1 Introduction

Ger is a G-code EmulatoR, G-code vERifier or a G-code transpilER (hence, "ger") intended for 3-axis CNC mills or routers. One a good day, you might pronounce it like your friend's name, Gary or Jerry; on a bad day, you might pronounce it as "grrrr."

Ger does two primary things. First, it translates more sophisticated G-code to a simplified form that should be acceptable to almost any CNC machine. Second, it displays an image of the object being created. Very few CNC controllers provide a true programming language, with variables, if-statements, loops, data structures, *etc.* Ger allows for "wizards," written in Java, to appear in the input code.

2 G-Code

There is no universally accepted standard for G-code. Many codes do adhere to common usage, but usage for the less common codes varies from one CNC controller to the next. Ger assumes that the common codes follow accepted usage, and does its best to finesse the other codes.

Ger is more flexible than many real-world controllers about the order and grouping of codes. For instance, something like this is accepted:

```
G01 X1.0 G21 G01 Y250.0
```

and would be interpreted as

```
G01 X1.0
G21
G01 Y250.0
```

Statements may be terminated by either a new-line (*i.e.*, a "carriage return" or "enter"), or by a semi-colon. Thus,

```
G01 X1.0; Y2.0
```

is valid and means something entirely different than

```
G01 X1.0 Y2.0
```

Anything that appears within parentheses, `(like this)` is treated as a comment. Ger allows multi-line comments, but not nested comments; the `'('` character may not be used within comments.

If it is possible for *you* to make sense of your program by reading the codes one chunk at a time, then Ger can probably make sense of it too, although the controller of an actual machine may be more strict or may have a slightly different interpretation.

Which leads to the next point. Ger is both a simulator and a translator. As a simulator, it can be used to view a 3D rendering of the part generated by the G-code, or it can be used to translate complex or idiosyncratic code to something simpler that will be accepted by a wide variety of machines. For example, not every controller accepts `M98` (call sub-program), even though it is fairly standard. Ger accepts `M98` and converts it to more basic commands that any real-world machine will accept.

Ger makes it possible to define your own functions that act as "wizards." For example, one might define

```
Pocket 75.0 30.0 10.0
```

to cut a pocket 10mm deep, measuring 75mm by 30mm. How to define these wizards will be discussed below.[1] In particular, wizard commands must start with two consecutive letters.

3 Input and Output Codes

To make the following tables more concise, `[...]` is used for an optional term, *n* means a number, possibly with a decimal, and *w* means a whole number.

3.1 Output Codes

The codes listed below may appear in the translated output, and they mean the same thing as both input and output. These codes are very common, and nearly any real-world CNC mill should interpret them the same way.

Move	`[X`*n*`] [Y`*n*`] [Z`*n*`] [F`*n*`]` This command moves the tool along a line segment to the given X, Y, Z, at the given feed rate. When in rapid mode (`G00`), `F` is not allowed. Once a feed rate is given, it will be used in `G01`-mode until the feed rate is changed. There is no particular code for "move;" whenever an `X`, `Y`, `Z`, or `F` appears, it is treated as a move command.
`G00`	Rapid travel mode. The program begins in rapid travel mode. If the very first line is `X1.000 Y1.000`, then the tool will move to that position in rapid mode.
`G01`	Ordinary (not rapid) travel mode.
`G02`	Circular interpolation, clockwise arc.

`G03`	Circular interpolation, counter-clockwise arc.
	`G02/G03 [X`*n*`] [Y`*n*`] [Z`*n*`] [I`*n*`] [J`*n*`] [K`*n*`] [F`*n*`] or`
	`G02/G03 [X`*n*`] [Y`*n*`] [Z`*n*`] [R`*n*`] [F`*n*`]`
	If an arc is specified using `R` rather than `I/J/K`, then a positive `R`-value produces an arc that subtends less than 180°; using a negative `R`-value produces an arc that's more than 180°.[2]

[1] WHERE?

[2] Peter Smid, *CNC Progrmming Handbook*, 2008 (3rd ed.), p. 253.

	To cut a complete circle, the X/Y/Z values given with the command must be equal to the starting point of the tool, and you must use the I/J/K format, not an R-value.
`G17`	Work in the XY-plane (the default).
`G18`	Work in the ZX-plane.
`G19`	Work in the YZ-plane. These commands (`G17/18/19`) set the reference plane for helical milling. The idea is that one of the coordinates is the odd man out; normally (in the XY-plane/`G17`) this extra coordinate is Z. In that case, any Z-value given with a `G02/03` produces a linear move in Z, so that the overall move is helical. Be aware that certain controllers may treat the ZX plane slightly differently, depending on whether they are "standard" or "vertical machining."[3]

That's it for the commands that actually move the cutter. The following items are commands that GER accepts and that pass through the translation process unchanged.

`M00`	Pause
`M01`	Pause. `M01` and `M02` are both forms of "temporary pause." Exactly how they are implemented varies depending on the real-world controller.
`M03`	Spindle on, clockwise. `M03 S`*w* where *w* is the spindle speed.
`M04`	Spindle on, counter-clockwise.
`M05`	Spindle off.
`M06`	Tool change. `M06 T`*w* changes to the tool in position *w* of the turret.
`M07`	Coolant on.
`M08`	Coolant on.
`M09`	Coolant off.
`M40`	Spindle high.
`M41`	Spindle low.
`M48`	Enable feed & speed overrides.
`M49`	Disable overrides.
`G43`	Positive tool length offset (TLO).
`G44`	Negative tool length offset. `G43/G44 [Z`*n*`] H`*w*

[3]Smid, p. 280.

	Ger considers the `Z`-value to be optional, although it is required by many real-world machines. If a Z-value is given, then the value is adjusted by `G20/21` (see below).
`G49`	Cancel tool length offset. The TLO commands are used on many machines to adjust for the fact that, when tools are stored in a turret, the location of the cutting edges of the various tools will differ. TLO can also be used to take cutter wear into account.

With one exception, the commands above are irrelevant unless the program is being executed on a real-world machine. The exception is `M06` for tool changes.

3.2 Other Accepted Codes

The codes described below are accepted by Ger, but are translated to the simpler codes listed above. Translation happens in stages, and the codes are presented below with all the codes that are digested and "translated away" in a single stage grouped together.

3.2.1 Program and Sub-program Codes

`M30`	End program.
`M98`	Call subprogram. `M98 P`*w* `L`*w* This calls a sub-program, where `P` gives its number, and `L` gives the number of times to call it.
`M99`	Return from sub-program.
`O`*w*	Programs and sub-programs must have an associated program number.

3.2.2 Unit Codes

The simulated machine works internally with inches or mm, and all codes will be output using those units by the translation process. The codes input to the translator may be given using either units, and these two codes indicate which one is being used.

`G20`	Inches.
`G21`	Millimeters.

3.2.3 Work Offset Codes

Ger translates all moves to be absolute, relative to machine zero. These commands change the PRZ; some of them refer to the work-offsets table. None of these codes may be used while in polar coordinates mode (`G16`), incremental mode (`G91`) or cutter comp mode (`G41/42`).

Most physical machines have a work offsets table that's used with the `G54-G59` commands. Work offsets are often used as a way to partition the tool table so that each partition has its own local coordinate system, with the code for different parts running more or less independently. This can be convenient in a production environment.

`G52`	`G52 [X`*n*`] [Y`*n*`] [Z`*n*`]` The PRZ is changed to be at the given X, Y, Z, relative to machine zero. If one of these values is omitted, then it is assumed to be zero.
`G54`	Apply entry 1 from the work-offsets table.
`G55`	Apply entry 2 from the work-offsets table.
`G56`	Apply entry 3 from the work-offsets table.
`G57`	Apply entry 4 from the work-offsets table.
`G58`	Apply entry 5 from the work-offsets table.
`G59`	Apply entry 6 from the work-offsets table. `G54` through `G59` are similar to `G52`, but the values used for X, Y and Z come from the work-offsets table.
`G92`	`G92 [X`*n*`] [Y`*n*`] [Z`*n*`]` The PRZ is changed so that the current cutter position has the given coordinates.

In theory, machine zero (or "home") is a position fixed at the factory.[4] Another form of origin is the Part (or Program) Reference Zero (PRZ). The PRZ is the coordinate system relative to which most commands are given. GER defines machine zero and the (initial) PRZ to be the location of the tool immediately before the first line of the program, so the tool always starts at (X,Y,Z) = (0,0,0).

Use `G52` to make the given values, expressed relative to machine zero, the new PRZ. Thus,

```
G52 X2.000 Y-4.000 Z1.00
```

means that the location 2 units to the right, 4 units below and 1 unit higher than machine zero will be treated as the PRZ in the code that follows. Saying

```
G52 X0 Y0 Z0
```

(or simply `G52`, with no arguments) resets the coordinate system to what it was when the program started.

[4]See Smid, p. 153, where he says that machine zero

> ...is the position of all machine slides at one of the extreme travel limits of each axis. Its exact position is determined by the machine manufacturer and is not normally changed during the machine working life.

In my experience that is not always true, but it's the idea reflected by machine zero.

Often, `G92` is the more useful command.[5] The values given with `G92` become the current coordinates under a new PRZ. For example,

```
G92 X1.000 Y-1.000 Z2.000
```

resets the PRZ so that the cutter's current location is given by $(1, -1, 2)$. Saying

```
G92 X0 Y0 Z0
```

(or simply `G92`, with no arguments) is a convenient way to make the current tool location the new PRZ.

The `G54-G59` commands work in a manner similar to `G52`, except that the arguments come from the work offsets table. That is, `G54` (no arguments) is the same as

```
G52 Xx Yy Zz
```

where `x, y` and `z` are taken from the work offsets table.

The important point is that GER will produce code with all coordinates given (in absolute terms) using the location of the tool at the start of the first line of the program as the orgin. To run the output code on a physical machine, the easiest thing do will *usually* be to set the PRZ to the surface of the lower-left corner of the part. As always, consider the possibility of interference and tool crashes.

3.2.4 Polar and Cartesian Coordinates

This isn't the place to explain how to use polar coordinates; see a math textbook.

`G15`	Polar coordinates off.
`G16`	Polar coordinates on.

Polar coordinates may only be used while in incremental mode (`G91`), and `G02/03` (arcs) are not permitted while in polar coordinate mode, nor may the PRZ be changed while in polar coordinate mode.

3.2.5 Absolute and Incremental Mode

Any move expressed in incremental terms is translated to absolute terms.

`G90`	Absolute mode.
`G91`	Incremental mode.

[5] This definition of `G92` is not universal. Some FANUC machines work this way (mine does), and other machines don't.

3.2.6 Cutter Comp Mode

This is often called “cutter radius compensation” mode.

This has not yet been implemented, so disregard this section.[6]

`G40`	Cancel cutter comp.
`G41`	Cutter comp, left.
`G42`	Cutter comp, right.

`G41/G42 D`*w or*
`G41/G42 H`*w*

The cutter comp is given relative to a value stored in a D-register or an H-register. The D-register typically contains the diameter of the tool and the H-register contains a value for tool wear. These are provided by using the “Tool Turret” dialog. As a reminder, the left/right distinction here refers to the position of the tool relative to its path. Thus, `G41` puts the tool to the left of the path it follows.

Ger does things slightly differently than how many real-world controllers seem to work. As soon as a `G41/42` appears, the virtual machine peeks ahead to see what the next statement will be and it bumps the tool out by the tool radius to be ready to cut the very first curve with the correct cutter comp. I think this is better than the way it’s typically done, but it could confuse people who are trying to test against a particular real-world machine. There seems to be so much variation from one controller to another in how this aspect of cutter comp is handled, that some degree of mis-match between Ger and the real world is unavoidable. In any case, the usual advice applies to *all* machines: invoke cutter comp away from the part and make at least one move before contacting the part.

Note that Ger does not allow things like

```
G41 X## Y## Z## D##
```

You can’t have any extra stuff between the `G41` and the `D` or `H`-register specification. Many real-world CNC machines do allow this.

[6]In fact, what I did earlier should be changed. I was trying to implement things so that the expectations of different machines could all be met, but that’s too confusing. Get rid of the whole H/D register aspect and just take the cutter radius from the tool table. Most machines do work that way, even if a few use the extra registers. Basically, give up on trying to emulate actual machines. That’s pointless.

One last point about cutter comp. Ger allows cutter comp to be used on interior angles, while many real-world controllers do not. However, Ger is not as smart as some real-world controllers about how it deals with this case. For instance, if the tool path is specified to form a very acute vertex, say $5°$, and the tool makes many small moves (less than the tool radius) near this vertex, then the tool path in cutter comp mode might not be what you expect. On the other hand, it's hard to imagine what a person might intend in a case like this, so it seems better to let the tool do oddball things as a way of informing him that he *input* something oddball.
About the only sitation I can imagine where this would arise is if you feed Ger code that was posted by a (rather stupid) CAM program. The way to "fix" this is to have infinite look-ahead in cutter comp mode so that Ger can remove all of those small moves near the apex from the statement stream, but it's not clear to me that this is something that *should* be fixed. It seems better for the controller to behave in a way that makes it apparent that the input G-code is weird, and probably wrong, rather than acting in a way that hides from the user potential problems with his G-code.

3.3 Omitted Codes

There are a few codes that some users might expect, but they are not accepted by GER.

`M02`	On modern machines, this is usually the same as M30 (end program). Back when machines used paper tape for input, this acted as a "stop," but did not rewind the tape.
`M47`	Usually, this means "repeat program," but it can mean other things – on some machines it's used for engraving (of all things).
`G28`	This typically moves the cutter to "home," and this action isn't typically needed outside the setup of a physical machine. GER defines machine zero to be $(0, 0, 0)$ and it's easy enough to move there with an ordinary move.
`N`	Some controllers allow a line to start with an `N`-value (for line number). GER does not.

In addition, GER does not provide any of the "canned cycles" available on many real-world machines. There is too much variety from one controller to the next, and wizards are a far more flexible approach.

EVERYTHING THAT FOLLOWS CAN BE IGNORED. IT NEEDS EXTENSIVE EDITING, AND MUCH OF IT IS WRONG OR OUT OF DATE.

4 Transpiling

Ger woks by transpiling (really, simplifying) the input code to the simplest possible subset of valid G-codes. This is done through a series of layers. When errors occur, it may be helpful to understand how the process of code simplification happens.

The first step is the lexical analizer, or "lexer". The lexer converts the input text to a sequence of "tokens." These tokens are of one of a few types. Ordinary G-code appears as a combination of letters, optionally followed by numbers. Examples include `G0`, `M30`, X3.205 and the like. The second type of token is associated with externally defined wizards. These include the function name, like `MyFunction`, and any arguments to such functions. These areguments are limited to numbers and strings delimited by a pair of double-quote (`"`) characters. The arguments to a wizard are space-delimited, and it is assumed that the arguments run to the end of the line on which the wizard is called.

The next layer is the parser. It converts the output from the lexer to syntacially valid statements.

4.1 Old Discussion...

The first thing is to set the internal units used by the simulated machine. Say

```
SetUnits ["inch"|"mm"]
```

The default is `"inch"`. This indicates the units for things like part dimensions and work offsets.

Once the units have been set, there are two ways to determine the part dimensions and machine zero (PRZ). The simplest is to say

```
SetMargin <value>
```

where `<value>` is the marginal amount that appears beyond the extent of where the tool cuts. In this case, the tool is assumed to touch the billet when the z-position drops below zero.

Another way to set the part dimension is to first say

```
SetPart <x> <y> <z>
```

where the values are the size of the raw (cubic) billet in the three dimensions. By default, machine zero (PRZ) is the top-lower-left of this material. The PRZ can be changed by saying

```
SetZero <x> <y> <z>
```

where the values are relative to the top-lower-left corner. For example,

```
SetPart 4.0 2.0 1.0
```

means that the part is $4 \times 2 \times 1$ (x by y by z). Now

```
SetZero 0.25 0.25 0.10
```

means that the PRZ is 0.25 in from the lower-left corner in both directions, and lowering the tool to Z$= 0$ cuts off the top 0.10 of the billet.

5 Wizards

THIS IS MORE OF A PLACEHOLDER...

The easiest way to develop a new wizard is through the command-line. Say (something like)

```
ger -compile mywizard.java
```

The only reason to do this, and not use `javac` is to avoid understanding the Java development framework any more than absolutely necessary.

You might also say

```
ger -translate gcode.txt
```

where `gcode.txt` uses `mywizard`. The issue here is that setting things up to allow GER to permit interactive Java development is hard for at least two reasons. First, is simply the UI – GER can't be an IDE. Second, is that it is hard to reload a given `.class` files. As the user changes his `.java` file, the `.class` file will change, and it's hard to reload changes to a class. The JVM doesn't understand that the current `MyWizard` class is different than the `MyWizard` it loaded earlier. It's possible to do this (IDEs do it), but a lot of messing around.

Part I
Old Java Version

6 System Overview

When the program is launched a window comes up with several menu choices. The “File” menu acts as you would expect. You can have multiple G-code programs open at the same time; each one appears in its own tab. You can edit and save programs from here too, although the editor is nothing fancy.

The “Render” menu generates a picture of the object described by the G-code program. There are two choices: “2D Display” and “3D Display”. Either choice generates a picture for the G-code in the foremost tab. The picture will appear in a new window, and the image is not automatically updated when you change the G-code. If you modify the G-code, and you want to see the result, choose “Render” again. Images for both versions of the G-code (new and old) will remain open. These images (and the data used to generate them) take a lot of memory. If you have several of these image windows open, then you could easily run out of memory.

Choosing “2D Display” shows a static image of the object, looking from above with each point shaded by the depth of cut. There is no user interaction (although you can make the window larger or smaller). If you choose “3D Display”, then you get an interactive 3D image of the object. You can’t use the mouse to rotate the image. Instead, use J and L to rotate left and right, I and M to rotate up and down, and U and O to twist the object in place. Use A and Z to zoom in and out. Also, X will close the window. Upper and lower case letters both work.

The “Settings” menu has several choices. Most of the choices should be self-explanatory, but not everything works. Both “Scale” and “Inches/mm” do work. If memory is tight, or if the image updates uncomfortably slowly, then reducing the scale should help. If the scale is s, then the virtual machine should work exactly as though the tool moves in steps of $1/s$ inches (or millimeters). Put another way, it’s as though you multiplied all of the coordinates in your program by a constant. For instance, if you change the scale from $1,000$ to 100, that’s like multiplying all coordinates by 0.10. For objects that are only a few inches in size and are of modest complexity, leaving the scale at 1000.0 should be fine, but for large objects, you may need to make this smaller. In fact, unless you’re doing something really intricate, a value of 100 (or even less) is probably fine.

The “Work Offsets” dialog does work (although it hasn’t been very thoroughly tested), and it should be clear what to do.

Don’t try to use the “Uncut Shape” dialog. It’s meant as a way for the user to indicate the size of the raw material, but it doesn’t work. Right now, the program will automatically use an object that’s large enough to encompass all cuts made by the program.

The "Tool Table" works, but only up to a point. The system recognizes *D* and *H* values from this table when using cutter comp and tool length offsets, and it knows the diameter of the tool. What it can't do is use anything other than simple endmills. If you change the diameter value in the *D*-column, then you must hit the return/enter key to tell dialog that you mean it before you click the "OK" button. One last thing: there's a secret default tool. This tool has a cutting diameter of 0.020 inches, but the cutter comp is 0.250 inches. This is handy sometimes when you're trying to understand what cutter comp is doing; it makes it easier to visualize the actual path of the tool. If there's no `M06` in your program, then you'll get the secret default tool.

Concerning Java and memory...When you launch the program that I wrote, your computer isn't really running my program. The computer runs Java, which in turn opens up and runs my program. This means that, whenever my program needs more memory, it must ask the Java system for the memory it needs. You may have oodles of memory on your machine, but Java won't let my program use it unless you tell Java that it's OK. I provided one way around this problem (a `.bat` file), but there are others.

7 G-codes and M-codes

I tried to implement all of the G-codes that one might care about and that are likely to be consistent across different makes of controller. Here's a list, with some comments.

`G40`	Cancel cutter comp.
`G41`	Cutter comp, left.
`G42`	Cutter comp, right. These work as they should, but (as usual) they might be wrong under `G18` and `G19`. Also, I do things differently than how most real CNC machines seem to work. As soon as a `G41/42` appears, the virtual machine peeks ahead to see what the next statement will be, and it bumps the tool out by the cutter comp amount to be ready to cut the very first curve with the correct cutter comp. I think this is better than the way it's typically done, but it could confuse people who are trying to test against a particular real-world machine. Anyway, there seems to be so much variation from one machine to another in how this aspect of cutter comp is handled, that some degree of mismatch between my virtual machine and the real world is unavoidable.

Dealing with cutter comp is messy and it's the thing most likely to have bugs. Also I chose to handle cutter comp by using what are probably very unorthodox methods. They are powerful, in the sense that I could extend my program to handle cutter comp on (as yet unspecified) curves that are more complex than line segments and arcs of circles, but there could be bugs for reasons that are hard to track down.

As you can see, none of the canned cycles, like `G81` for drilling, have been implemented. Some of them might manage to get through my system without causing an error, but none of them do anything. I haven't thought very hard about this, but I don't think that they would be hard to add.

8 Bonus

The "Test" menu is there for my own debugging purposes. It's not meant to be very clean, but there are a few things there that you might think are interesting.

The output of "Test Voxel Frame" looks something like what you see in the 3D window under Mach3. Every tool position gets a dot in three-dimensional space, and this draws those dots. The drawing is cruddy, but it wasn't really meant for public consumption. You can rotate and zoom using the keyboard.

Choosing "Test Pulses" will show the series of pulses (zeros and ones) generated by the G-code. These are the signals that would be sent to the motors of a real-world CNC machine. Again, this was for debugging, and I think that it only shows the first 100 pulses.

The other choices under the "Test" menu have to do with the way that my program converts G-code to pulses. It's a multi-step process that begins with the lexer (in computer science lingo, the "lexical analyzer"). The lexer converts the string of characters that make up the G-code into bite-sized pieces that later stages find easier to digest. For instance, the lexer converts

```
X0.100 Y1.500
```

into two "tokens", one for the X-value and another for the Y-value. That's probably not so interesting to you, but the other layers of the process might be.

The parser takes the tokens from the lexer and converts them to meaningful statements. Basically it verifies that the G-code isn't absolute gibberish. The output of "Parsify" should look pretty much like the input G-code.

The next five steps, "Simplify 00" through "Simplify 05" take the original G-code and convert it to G-code of a simpler and simpler form. The 00 step eliminates subroutine calls. The output is exactly what you would have to type (over and over again) if there were no such thing as `M98/99`. The 01 step converts everything to the standard unit of the machine (inches or millimeters) so that `G20/21` no longer appear in the G-code. The 02 step removes any work offsets so that `G52` and `G54` through `G59` are eliminated, along with `G43,44,49` for tool length offset. All coordinate values generated by step `01` are adjusted as needed. The 03 step gets rid of polar coordinates. The 04 step changes all coordinate values to be absolute so that `G90/91` is not needed in the output, and step 05 adjusts everything for cutter comp.

The output of step 05 is exactly the G-code that you would have to type to create the same object as the original G-code, but on a machine that understands only `G00, G01, G02 and G03`, along with the `M`-codes for tool change and spindle & coolant control. The ability to see the output of this final step may be useful from a teaching point of view. It would help students to visualize what cutter comp and subroutine calls (for example) really do.

Part II
C++/Qt Version

9 Overview of Menus

When the program is launched a window comes up with several menu choices. The "File" and "Edit" menus act as you would expect. You can have multiple G-code programs open at the same time; each one appears in its own tab. You can edit and save programs from here too, although the editor is nothing fancy. You can adjust how much space each of the left and right panes has by dragging the divider between them.

Use the "Action" menu to perform a translation/simplification of your G-code or to show an image of the object. After you've loaded some G-code, either by opening a file or by typing it in, choose "Translate" and a simplified version of your program will appear in the right panel. The format is basically the same as ordinary G-code, although it's been restated to use only the most fundamental linear and arc moves. The line numbers that appear on the right (the N-values) refer to the line in the input file that lead to that line of output. You can use this to understand where your input program went wrong – why didn't the simulated controller move the tool in the way I thought my G-code specified?

The "Action" menu is where you can bring up a various dialog boxes that set up the simulated machine. The "Set Geometry" dialog allows you to define the dimensions of the raw material being machined, whether to work in inches or millimeters, the location of the PRZ (*i.e.*, the origin) and the tool turret.

The "Tool Table" dialog allows you to specify up to 20 tools in the turret. Ger is able to cut with endmills, ballmills (cutters with a hemispherical tip), center drills and ordinary drills. In fact, a "drill" is treated as if it were an ordinary pointy-ended mill; you can make linear cuts or plunging cuts with one. Programs change tools in the ordinary way, by using `M06`. For instance, `M06 T3` changes to the third tool in the turret. If a program doesn't specify a particular tool, then it defaults to the first tool in the turret, which is a quarter-inch endmill.

If your program invokes `G55/56/57/58/59`, then use the "Work Offsets" dialog to set the corresponding values.

Finally, the "Scale" dialog allows you to adjust the level of detail at which the program works. For most uses, it's not necessary for rendered images to be calculated to 0.001 in – at that level of detail a square inch of material would fill an unusually large computer monitor. If you do need more detail, the "Scale" dialog allows you to get it, although the rendering process may take up to 100 times longer and use 100 times the memory.

To show an image of the part cut by your G-code, choose "Render – 2D." If you modify the G-code, and you want to see the result, choose "Render – 2D" again.

10 G-codes and M-codes

`G40`	Cancel cutter comp, also known as "tool radius compensation."
`G41`	Cutter comp, left.
`G42`	Cutter comp, right.

`G41/G42 D`*w or*
`G41/G42 H`*w*

The cutter comp is given relative to a value stored in a D-register or an H-register. The D-register typically contains the diameter of the tool and the H-register contains a value for tool wear. These are provided by using the "Tool Turret" dialog. As a reminder, the left/right distinction here refers to the position of the tool relative to its path. Thus, `G41` puts the tool to the left of the path it follows.

Ger does things slightly differently than how many real-world controllers seem to work. As soon as a `G41/42` appears, the virtual machine peeks ahead to see what the next statement will be and it bumps the tool out by the tool radius to be ready to cut the very first curve with the correct cutter comp. I think this is better than the way it's typically done, but it could confuse people who are trying to test against a particular real-world machine. There seems to be so much variation from one controller to another in how this aspect of cutter comp is handled, that some degree of mis-match between Ger and the real world is unavoidable. In any case, the usual advice applies to *all* machines: invoke cutter comp away from the part and make at least one move before contacting the part.

Note that Ger does not allow things like

```
G41 X## Y## Z## D##
```

You can't have any extra stuff between the `G41` and the `D` or `H`-register specification. Many real-world CNC machines do allow this.

One last point about cutter comp. Ger allows cutter comp to be used on interior angles, while many real-world controllers do not. However, Ger is not as smart as some real-world controllers about how it deals with this case. For instance, if the tool path is specified to form a very acute vertex, say 5°, and the tool makes many small moves (less than the tool radius) near this vertex, then the tool path in cutter comp mode might not be what you expect. On the other hand, it's hard to imagine what a person might intend in a case like this, so it seems better to let the tool do oddball things as a way of informing him that he *input* something oddball.

About the only sitation I can imagine where this would arise is if you feed Ger code that was posted by a (rather stupid) CAM program. The way to “fix” this is to have infinite look-ahead in cutter comp mode so that Ger can remove all of those small moves near the apex from the statement stream, but it’s not clear to me that this is something that *should* be fixed. It seems better for the controller to behave in a way that makes it apparent that the input G-code is weird, and probably wrong, rather than acting in a way that hides from the user potential problems with his G-code.