1 Yet to Do

There's a long list of things wrong with this. It's semi-usable, but here's a list of things to revisit (or visit).

- Allow user to specify font size (and maybe the font, although any sane person will want a mono-spaced font).
- The basic framework for wizard code is there, but no more than the framework. In particular, the means of outputing more types of `Statement` needs to be fleshed out.
- Certain wizards (like Bezier curves) should have a UI specific to them. Other UIs might be handy too (bolt circles), but are less necessary. None of these wizards exist yet in any form.
- The "machine setup" wizard framework is also minimal. This is for things like specifying the tool table, work offsets table, billet dimensions, etc.
- Go through the code and explain (in this document) how each transformation takes place. There are edge cases that might surprise the user. Some of the more detailed calculations belong in the developer's manual.
- There is no 3D rendering at all.

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 more problematic 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
```

If it is possible for *you* to make sense of your program by reading the codes one chunk at a time, then the simulator can probably make sense of it too, although the controller of an actual machine may be more strict or may have a 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 (WHERE?). In particular, wizard commands must start with two consecutive letters.

3 Common Codes

The codes described below are common enough to assume that they take a standard form and have a standard meeaning. For any codes not on this list, GER assumes that the full statement runs to the end-of-line or the next semi-colon.

The table below is organized by "statement," meaning a series of inputs that makes sense together. To make the descriptions more concise, `[...]` is used for an optional term, *n* means a number, possibly with a decimal, and *w* means a whole number. What appears below is a quick summary; the individual codes are described in more detail later.

Move	`[X`*n*`] [Y`*n*`] [Z`*n*`] [F`*n*`]` Notice the use of `[` and `]`: each of these terms is optional. 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.
`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`	Clockwise arc.
`G03`	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*`]`

Circular interpolation. 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°. See Smid, p. 253.

Also, various machines may treat the ZX plane slightly differently, depending on whether they are "standard" or "vertical machining." So, `G02` and `G03` may be treated in opposite fashion. Basically, what does "clockwise" mean? See Smid, *CNC Programming Handbook*, p. 280.

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.

Geometrically, the distance of the center to each of the two end-points must be the same (that distance is the radius). When using the I/J/K format, it's not unusual for the center of an arc to require many digits to specify exactly, perhaps even an infinite number. So that the programmer only needs to input a reasonable number of digits, a certain amount of leeway must be allowed. Ger will accept a center if the distances to the two end-points are within 1% of each other, and it will assume that the radius of the circle is equal to the larger of these two distances.

`G15` Polar coordinates off.

`G16` Polar coordinates on.

I'VE CHANGED THIS. Really, polar coordinates are inherently an incremental concept. If you wanted to use them in absolute mode, the only reasonable way to do so would be relative to machine zero, and that would be strange.

You may enter polar coordinate mode while in incremental mode (`G91`) or you may enter incremental mode after invoking `G16`, or go back to absolute mode (`G90`). Not every real-world CNC controller allows this. Ger also allows circular interpolation (`G02/G03`) while in polar coordinate mode, another thing that most real-world machines do not permit. Use of `G01` or `G00` while in polar coordinate mode is *not* allowed.

`G17` Work in the XY-plane (the default).

`G18` Work in the ZX-plane.

`G19` Work in the YZ-plane.

While in cutter comp mode, only the standard XY-plane may be used. This is typical for real-world controllers.

I NEED TO EXPLAIN A LITTLE ABOUT WHY THAT IS.

`G20` Inches.

`G21` Millimeters.

Ger works internally using either inches of millimeters, but you can freely change from one unit of measure to the other within your G-code. In fact, Ger is more flexible than many real-world controllers; you can sprinkle `G20`'s and `G21`'s all over the place. The only (?) reason the internal setting for GER matters is how it interprets things like tool tables.

`G28` Machine Zero Return. This is accepted, but everyting from `G28` to the end of the line (or a semi-colon) is ignored. On the simulated machine, it has the effect of moving the tool to the position of the tool turret.[1]

`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.

[1]In fact, I'm not sure whether I've implemented 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.

`G43` Positive tool length offset (TLO).

`G44` Negative tool length offset.

`G49` Cancel tool length offset.

`G43/G44 [Z`*n*`] H`*w*

The simulator considers the `Z`-value to be optional, although it is required by many real-world machines.

As a reminder, in the case of `G43`, this has the effect of adding the value in the H-register to all Z-moves until a `G49` is reached. If the H-value is negative, then the tool will cut more deeply. `G44` has the opposite effect.

A program may not enter TLO while in incremental mode, when using polar coordinates, or while in cutter comp mode. However, once TLO is in effect, a program may enter any of these modes. If your program does enter one of these modes, then it must leave these modes before cancelling TLO.

`G52` Local coordinate system.

`G52 [X`*n*`] [Y`*n*`] [Z`*n*`]`

This changes the origin to be the point at the given X, Y, Z, stated relative to the PRZ. All moves after `G52` are as though the PRZ is at this new position. Each of the X, Y, and Z values is optional; omitted values are assumed to be zero (no change). This is cancelled with `G54`.

`G54` Return to normal work offset. That is, use the PRZ as the origin. If a tool length offset is in force (`G43` or `G44`), then it remains in force after calling `G54`.

`G55`	
`G56`	
`G57`	
`G58`	
`G59`	Codes `G54` through `G59` are similar to `G52`, but the values used for X, Y and Z come from the “work offsets” table. So, these codes appear alone, simply as `G55` (say), without any additional codes. Just as with TLO mode, none of these codes, including `G52`, may be entered or exited while in incremental mode, polar coordinate mode, or cutter comp mode. Normally, the values used for work offsets are the distance from machine zero to program zero (the PRZ), but the virtual machine has no “machine zero.” For the purpose of work offset settings, think of the PRZ as the machine zero. Put another way, the work offset values should be the amount in each direction, X, Y and Z, that you want the tool to be offset from the position it would have without the change due to applying one of `G55` through `G59`.
`G90`	Absolute mode.
`G91`	Incremental mode.

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.

Here are the valid M-codes.

`M00`	
`M01`	
`M02`	These are different forms of “program stop.” The simulator simply halts when it reaches one of these.
`M03`	Spindle on, clockwise. `M03 S`*w* where *w* is the spindle speed.
`M04`	Spindle on, counter-clockwise. Similar to `M03`. The program notes internally that the spindle is on, but it has no real effect.
`M05`	Spindle off.
`M06`	Tool change. `M06 T`*w* changes to the tool in the *w* position of the turret. The simulator moves the tool in a straight line from its curent position to the tool turret, changes to the new tool, and stops at that position.[2]
`M07`	

[2]These motions don’t appear in the translator part of the program; they are used to detect tool crashes.

`M08`	
`M09`	These coolant on/off commands are accepted, but they do nothing.
`M30`	Halts the program.
`M40`	
`M41`	Spindle high/low. Accepted, but ignored.
`M47`	Repeat program. The simulator halts immediately if it reaches this code.
`M48`	
`M49`	Feed & speed override on/off. Accepted, but ignored.
`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 subprogram.

Two other codes are accepted: the `O` code for a program number, and `N` may be used for line numbers. Any line numbers given with `N` are ignored. Every program is required to start with an O-value.

If there are any errors in the program, then the line number on which the errors are detected are given by counting lines in the input text file, not with reference to any `N`-specified line numbers in the input G-code. Thus, if an error is reported on line 3 (say), then the problem was detected on the third line from the top, not the line that starts with `N3` (assuming there is such a line).

Finally, comments are expressed with parenthesis, `(...)`, as usual. The simulator accepts multi-line comments, but does not accept nested comments.

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.

5 Coordinate Systems

The coordinate system is inherently tied up with the particular physical machine on which the G-code is intended to run. Codes like `G28` (machine zero return) are impossible to interpret in a way that would be consistent with every machine. Some other codes related to the coordinate system are accepted by GER, but "translated away."

Machines typically have several notions of coordinate system origin. In theory, "machine zero" (or "home") is a position fixed at the factory[3] and `G28` should return the cutter to that position. 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. As with most real machines, the location of machine zero can't be changed. So, the tool always starts at (X,Y,Z) = (0,0,0). GER *does not accept the* `G28` `command`; there's very little reason to return the tool to (0,0,0), and it's always possible to move there with `G00` or `G01` in the usual way.

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.

5.1 Changing the PRZ

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
```

[3]See Peter Smid, *CNC Progrmming Handbook*, 2008 (3rd ed.), 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. In any case, it would be impossible for GER to reflect the quirks of every possible machine.

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

Often, `G92` is the more useful command.[4] 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.

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 in each partition. This can be handy in a production environment. Ger has a work offets table, even though it seems less useful in a translator.

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.

5.2 Tool Length Offset

The `G40`, `G43` and `G49` commands are used on many machines for tool length offset (TLO). This is used 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.

Ger accepts these commands, but they pass through unchanged, without affecting the surrounding code. In theory, Ger could have been written to simulate the effect of the TLO commands in more detail, but doing so would require that the tool table reflect the irritating real-world quirks of various cutters and turrets. The value of Ger's translation and simulation is that it allows you to avoid these irritations – or at least, to avert your eyes until they can't be avoided.

[4]This definition of `G92` is not universal. Some FANUC machines work this way, and other machines don't.

5.3 Old Discussion...

Exactly how to define the geometry of the machine and the part, together with various concepts of "origin," is a difficult question. On the other hand, it's often the case that certain obvious defaults will suffice. To fully define the geometry of the machine, follow these steps:

1. Define the size of the tool table.
2. Set the location of the tool turret relative to the lower-left corner of the tool table. The turret location is the position to which the spindle will return for a tool change.
3. Define the dimensions of the billet of material. By default, it is assumed to be centered on the tool table.
4. Set the PRZ relative to the billet. By default, this is taken to be at the lower-left corner of the billet.

The series of steps above is fussy and not necessary in most cases. Usually, it suffices to work with default settings. In this case, the billet is taken to be just large enough for the cuts that appear in the G-code (should allow setting some kind of margin) and the PRZ ends up being at the lower-left of the material. Tool changes happen in-place, without a move to a particular location for the tool turret. Things like wook tables might be trickier though.

THIS WON'T WORK. For one thing, the tool table moves on some machines. Other machines might have the tool table move in only X or Y. It's just too fiddly. What I need to do is break the list of codes up into the obvious ones, and the ones that are somehow sensitive to the machine geometry. The commands that are somehow sensitive to the geometry are `G28` (machine zero return), `G43, G44, G49` (to do with tool length offset), `G52-G59` (work offsets), `M06` (tool change)

For things that have to so with the coordiante system, it's reasonable to make everything relative to a single PRZ, which can default to the lower-left corner of the billet, perhaps with some margin.

I will ignore the location of the tool changer, and just have the tool magically change in place. The user could redefine `M06` somehow though to have it move to a certain location.

That leaves the tool length offset commands, and those aren't a big deal.

Overall, the best approach seem to be to specify all of this stuff within the G-code file. Each program could have different settings, and it's misleading to use the main window to do this. What could be done in the main window, as a kind of machine-wide setting, is the tool table. In any case, this is a more complicated thing that could be difficult to do with simple text.

So, what's needed is the following.

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.

6 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

7 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.

8 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.

`G00`	Rapid travel.
`G01`	Ordinary move. There are no surprises with these two. However, for all practical purposes, my program makes no distinction between `G00` and `G01`. The program does not check for things like whether the spindle is on when you cut, or whether you're trying to cut in rapid mode. Likewise, the program accepts a feed rate (F-value), but the feed rate has no effect on the output.
`G02`	
`G03`	Circular interpolation. These do what you'd expect, although I'm not very confident that they work correctly if you are not in `G17` mode.
`G04`	Dwell. This is accepted, but ignored. Actually, it might cause an error, depending on how you use it. I'm not sure.
`G15`	Polar coordinates off.

`G16`	Polar coordinates on. Again, using `G18` and `G19` might not work right. I tried to make them work correctly, but it hasn't been thoroughly tested.
`G17`	
`G18`	
`G19`	See above and below for caveats about `G18` and `G19`.
`G20`	Inches.
`G21`	Millimeters. The abstract machine works internally in either inches of millimeters, but you can freely change from one unit of measure to the other within your G-code. In fact, my program is probably more flexible than many real controllers, perhaps dangerously so. You can sprinkle `G20`'s and `G21`'s all over the place.
`G28`	This is accepted, but ignored. Smid calls it "machine zero return"; I'm not sure exactly what it should do.
`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.
`G43`	Tool length offset, in the normal case.
`G44`	Tool length offset, in the negative sense. I doubt that this is used often, but it works.
`G49`	Cancel tool length offset. Tool length offset works as it should. One oddity is due to the fact that, in a virtual machine, tools don't have a natural length. For the time being, treat all tools as though the machine magically knows where the tip of every tool is. If you use `G43` after setting $H = 0$ in the tool table, then it will act like it should. If $H > 0$, then the tool will cut more deeply under `G43` and less deeply under `G44`.

`G52`	Allows you to specify a new local reference frame. This is something like a work offset (`G54`,...,`G59`), but the coordinates are specified in the program, and must be given relative to the PRZ.
`G54`	
`G55`	
`G56`	
`G57`	
`G58`	
`G59`	To change work offsets. These work as you would expect, but the coordinates must be given relative to the PRZ in the work offset table. There's no natural way to define "machine coordinates" other than the PRZ.
`G90`	Absolute mode.
`G91`	Incremental mode. These work as expected.

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.

Here are the valid M-codes.

`M01`	Smid calls this "compulsory program stop."
`M02`	And he calls this "optional program stop." In both cases, I make the program halt, like an `M30`.
`M03`	Spindle on, clockwise.
`M04`	Spindle on, counter-clockwise. The program notes internally that the spindle is on, but it has no real effect.
`M05`	Spindle off.
`M06`	Tool change.
`M07`	
`M08`	
`M09`	These coolant on/off commands are accepted, but they do nothing.
`M30`	Halts the program.
`M40`	
`M41`	Spindle high/low. Accepted, but ignored.
`M47`	Repeat program. Since I'm not sure exactly what this should do, it generates an error.
`M48`	
`M49`	Feed & speed override on/off. Accepted, but ignored.
`M98`	Call subprogram.
`M99`	Return from subprogram. Both of these work.

The system also accepts line numbers (with `N`), but they don't have any meaning since none of the valid codes can refer to line numbers.

9 Shortcomings

The most obvious shortcoming is that it can be slow to bring up the image. Once the image is up, it's reasonably fast to move around in 3D. The larger the tool is, the slower it is. If the tool is only 0.020 in diameter, then it's plenty fast, but a tool that's 0.250 in diameter takes roughly 150 times as long as a tool that's only 0.020 in diameter. Having to wait 10 seconds, or even a minute for the image to come up is possible. In round numbers, with a quarter inch endmill, you can expect to wait 0.25 seconds for every inch of tool travel (at least that's true on my machine). I have ideas about how to fix this, but it will take some time to make the necessary changes. As a quick fix, you can change the scale value from 1,000 to something like 100.

The other big problem is the fact that the program is a memory hog. There are things I can do to improve matters, but some amount of hoggishness is unavoidable. The program isn't very nice about its memory needs; it may crash if it can't get the memory it wants.

You'll notice that, in 3D mode, the image you get isn't really a solid object at all; it's just the "skin" of the object's top. Making the object appear as a solid is easy, but I find that, given the less than ideal method of shading that I use now, it's easier to visualize what's been cut if the object isn't solid.

Objects that are large and/or intricate will be drawn slowly in 3D mode. You'll notice this for anything larger than about three inches square. I mentioned above that you can reduce the scale value as a crude fix for this problem. Doing that makes the internal representation of the object less detailed. I have ideas about how I can improve the speed, but implementing these idea will take some time.

The way the object is shaded isn't the greatest. Right now, the surface of the object is shaded based on the depth of the cut. This is OK most of the time, but sometimes it would be nice if there were more contrast between adjacent surfaces. I know how to fix this (I think), but it's non-trivial.

If you zoom in too far on objects, the program may crash. I know exactly why this happens, but I'm probably going to totally redo that part of the program anyway, so I don't want to deal with the problem.

Even for very simple objects, there's an internal limit on the permitted size of the object. This limit is roughly a cube 32 inches on each side. I could allow objects that are literally miles to the side by doubling the memory requirements, but, unfortunately, there's no middle ground. In any case, my current less efficient methods of memory usage dictate a limit of roughly a square six inches on a side.

10 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

11 What is This Thing?

For the time being, I am calling this program "Ger," which I got by combining "G," as in "G-code," with the "er" sound of "verify." So, the program is a G-code verifier or a "Gerifier" – "Ger" for short. You can pronounce as in the name Gary (or Jerry) when you feel friendly toward the program, or a bear's growl when the program isn't doing what you want it to, which shouldn't be very often.

I thought of calling the program "Gim," as in "G-code simulator," but I'll bet that name is taken for something. If you think of a better name, let me know.

As you use this program, please let me know if you find anything about it that doesn't seem right. Obviously, if it translates your code in a way that doesn't make sense, I want to know about that. In fact, I'd like to hear about anything odd or unexpected that the program does. If you see an inefficiency, inelegance, or just have a good idea related to the program, then I want to hear about that too. If you do find any bugs, it will help me if you can send me the input G-code and/or explain (in detail!) what you did that led to the strange behavior. I can't fix a bug that I can't reproduce. I am at `rsfairman@yahoo.com`**.**

At this point the program can do two things: translating G-code to a simpler form and rendering a two-dimensional image of the part being cut.

11.1 Translation

You can give Ger some G-code, and have it spit out a simplified version of the code you gave it. Here's what I mean.

Fundamentally, any G-code program boils down to having the tool do one of two things: move linearly or move along a circular arc – there are also helical moves, but these are just a combination of a linear move and an arc move. For example, if the program you input involves cutter compensation (`G42`), work offsets (`G56`), polar coordinates (`G16`), incremental mode (`G91`), and so forth, then the output program will have the same effect as the input program – will move the tool in the same way and cut the same part – but it won't involve any of these more complicated commands. The output program will involve only `G00, G01, G02` and `G03`.

In fact, there are a few codes in addition to these four that can't be eliminated from the output program. These additional codes are `G17/18/19`, which specify which of the XY, ZX or YZ-plane to work in, along with the M-codes for spindle start/stop (`M03/04/05`) and tool change (`M06`). In a section below, there is a list of the G/M-codes that Ger accepts as input, along with a brief description of what each one does.

The translator is useful for several things. It's a good teaching tool. The more esoteric G-codes can be difficult to understand, and once you do understand them, it's easy to forget how they work. By running your G-code through Ger, and looking at the output, you can check your understanding of particular codes.

Second, not all controllers are very friendly about certain G-codes. For instance, I have had bad luck using cutter comp on Mach3. Maybe it's me, but I've had Mach3 go haywire too many times when I was using cutter comp to trust it. Using Ger, you can write your program with cutter comp, have Ger translate it to a program which does the same thing, but does it without cutter comp, and give that simpler program to the real-life controller.

Third, even experts get confused. If you're writing a long program, particularly one that uses sub-programs and incremental mode, it's easy to get mixed up about where the tool is at a particular point in the program. Ger expresses its output strictly in absolute mode, making it easy to check that the tool is where you think it is as the program proceeds. If the tool is not where you think it should be, then Ger makes it easy to find out where your program went wrong.

11.2 2D Rendering

Ger is able to show a flat (2D) image of the object cut by your G-code (I'm working on 3D rendering, but it's not ready yet). The depth of cut is indicated by the shade of gray used, with darker areas being cut more deeply.

Once the image is visible, you can drag it around with the mouse or zoom in and out by using the mouse wheel. Double-click on the image to bring it back to the standard view showing the entire object centered in the window. You can use the keyboard instead of the mouse: use the arrow keys to pan the image, the plus and minus keys to zoom in and out, and the enter key to return to the standard view.

12 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.

13 G-codes and M-codes

Different controllers use slightly different variations of the G/M-codes. Ger tries to take a middle-of-the-road approach, although it 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`. If it is possible for *you* to make sense of your program by reading the codes one "chunk" at a time, then the simulator can probably make sense of it too, although the controller on the actual milling machine may be more strict.

Many of the codes that make sense on a real-world machine are ignored by the simulator. For instance, `M08, M09` (coolant on/off) is ignored, and `M47` (repeat program) doesn't really make sense in this context.

To be clear on how this program interprets the different codes, here's a list of how they are seen by the program's virtual controller. The table below is organized by "actionable chunks," meaning a series of inputs that makes sense together. To make the descriptions more concise, `[...]` is used for an optional term, *n* means a number, possibly with a decimal, and *w* means a whole number.

Move	`[X`*n*`] [Y`*n*`] [Z`*n*`] [F`*n*`]` Notice the use of `[` and `]`: each of these terms is optional. This command moves the tool along a line segment to the given X, Y, Z, at the given feed rate. If you are in rapid mode (`G00`), then the `F` is not allowed.
`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`	Clockwise arc.
`G03`	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*`]` Circular interpolation. These do what you'd expect. Something to remember: 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°. Also, various machines may treat the ZX plane slightly differently, depending on whether they are "standard" or "vertical machining." So, `G02` and `G03` may be treated in opposite fashion. Basically, what does "clockwise" mean? See Smid, *CNC Programming Handbook*, p. 280. 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.

Geometrically, the distance of the center to each of the two end-points must be the same (that distance is the radius). When using the I/J/K format, it's not unusual for the center of an arc to require many digits to specify exactly, perhaps even an infinite number. So that the programmer only needs to input a reasonable number of digits, a certain amount of leeway must be allowed. Ger will accept a center if the distances to the two end-points are within 1% of each other, and it will assume that the radius of the circle is equal to the larger of these two distances.

`G04` Dwell. This is accepted, but everything from `G04` to the end of the line (or a semi-colon) is ignored.

`G15` Polar coordinates off.

`G16` Polar coordinates on.

You may enter polar coordinate mode while in incremental mode (`G91`) or you may enter incremental mode after invoking `G16`, or go back to absolute mode (`G90`). Not every real-world CNC controller allows this. Ger also allows circular interpolation (`G02/G03`) while in polar coordinate mode, another thing that most real-world machines do not permit. Use of `G01` or `G00` while in polar coordinate mode is *not* allowed.

`G17` Work in the XY-plane (the default).

`G18` Work in the ZX-plane.

`G19` Work in the YZ-plane.

While in cutter comp mode, only the standard XY-plane may be used. This is typical for real-world controllers.

`G20` Inches.

`G21` Millimeters.

Ger works internally using either inches of millimeters, but you can freely change from one unit of measure to the other within your G-code. In fact, Ger is more flexible than many real-world controllers; you can sprinkle `G20`'s and `G21`'s all over the place.

`G28` Machine Zero Return. This is accepted, but everyting from `G28` to the end of the line (or a semi-colon) is ignored. On the simulated machine, it has the effect of moving the tool to the position of the tool turret.[5]

`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*

[5]In fact, I'm not sure whether I've implemented this.

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.

`G43`	Positive tool length offset (TLO).
`G44`	Negative tool length offset.
`G49`	Cancel tool length offset. `G43/G44 [Z`*n*`] H`*w* The simulator considers the `Z`-value to be optional, although it is required by many real-world machines. As a reminder, in the case of `G43`, this has the effect of adding the value in the H-register to all Z-moves until a `G49` is reached. If the H-value is negative, then the tool will cut more deeply. `G44` has the opposite effect. A program may not enter TLO while in incremental mode, when using polar coordinates, or while in cutter comp mode. However, once TLO is in effect, a program may enter any of these modes. If your program does enter one of these modes, then it must leave these modes before cancelling TLO.
`G52`	Local coordinate system. `G52 [X`*n*`] [Y`*n*`] [Z`*n*`]` This changes the origin to be the point at the given X, Y, Z, stated relative to the PRZ. All moves after `G52` are as though the PRZ is at this new position. Each of the X, Y, and Z values is optional; omitted values are assumed to be zero (no change). This is cancelled with `G54`.
`G54`	Return to normal work offset. That is, use the PRZ as the origin. If a tool length offset is in force (`G43` or `G44`), then it remains in force after calling `G54`.
`G55`	
`G56`	
`G57`	
`G58`	
`G59`	Codes `G54` through `G59` are similar to `G52`, but the values used for X, Y and Z come from the "work offsets" table. So, these codes appear alone, simply as `G55` (say), without any additional codes. Just as with TLO mode, none of these codes, including `G52`, may be entered or exited while in incremental mode, polar coordinate mode, or cutter comp mode. Normally, the values used for work offsets are the distance from machine zero to program zero (the PRZ), but the virtual machine has no "machine zero." For the purpose of work offset settings, think of the PRZ as the machine zero. Put another way, the work offset values should be the amount in each direction, X, Y and Z, that you want the tool to be offset from the position it would have without the change due to applying one of `G55` through `G59`.
`G90`	Absolute mode.
`G91`	Incremental mode.

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.

Here are the valid M-codes.

`M00`	
`M01`	
`M02`	These are different forms of "program stop." The simulator simply halts when it reaches one of these.
`M03`	Spindle on, clockwise. `M03 S`*w* where *w* is the spindle speed.
`M04`	Spindle on, counter-clockwise. Similar to `M03`. The program notes internally that the spindle is on, but it has no real effect.
`M05`	Spindle off.
`M06`	Tool change. `M06 T`*w* changes to the tool in the *w* position of the turret. The simulator moves the tool in a straight line from its curent position to the tool turret, changes to the new tool, and stops at that position.[6]
`M07`	
`M08`	
`M09`	These coolant on/off commands are accepted, but they do nothing.
`M30`	Halts the program.
`M40`	
`M41`	Spindle high/low. Accepted, but ignored.
`M47`	Repeat program. The simulator halts immediately if it reaches this code.
`M48`	
`M49`	Feed & speed override on/off. Accepted, but ignored.
`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 subprogram.

Two other codes are accepted: the `O` code for a program number, and `N` may be used for line numbers. Any line numbers given with `N` are ignored. Every program is required to start with an O-value.

If there are any errors in the program, then the line number on which the errors are detected are given by counting lines in the input text file, not with

[6]These motions don't appear in the translator part of the program; they are used to detect tool crashes.

reference to any `N`-specified line numbers in the input G-code. Thus, if an error is reported on line `3` (say), then the problem was detected on the third line from the top, not the line that starts with `N3` (assuming there is such a line).

Finally, comments are expressed with parenthesis, `(...)`, as usual. The simulator accepts multi-line comments, but does not accept nested comments.