Natural Language Control for XY Gantry Systems

Ian Ding
iding918@bu.edu

Wai Teng Sin
isaacsin@bu.edu

Abstract

In this paper we present a system for translating natural language instructions into G-code for XY gantry systems. We explore three neural architectures: a baseline RNN model, a custom transformer encoder–decoder model, and a StarCoder decoder-only model fine-tuned with Low-Rank Adaptation (LoRA) and 4-bit quantization. While the transformer and RNN baselines demonstrates limited usability, the fine-tuned StarCoder with LoRA significantly improved performance. This work represents a step towards democratizing robotics through natural language interfaces and lays the foundation for open-ended, precise, and user-friendly control of automated robotic platforms.

1 Introduction

XY–gantry platforms underpin a wide range of contemporary applications—from pick-and-place machines on factory floors, to high-throughput biomedical assays, to camera sliders used by content-creators. Yet interaction with these robots remains locked behind low-level machine languages such as *G-code*. While G-code offers deterministic control, its terse syntax and implicit assumptions about coordinate frames, feed-rates and modal states make it a barrier for makers, educators and small businesses who cannot afford dedicated automation engineers. The result is a yawning gap between the physical capability of the hardware and the skill set of end-users who would benefit most from it.

MakerMods[1] is a young robotics startup founded by Ryan Chan and Wai Teng Sin whose mission is to *democratize* hardware by fusing modular USB-C electronics (*ModBlocks*) with natural-language, AI-powered control. In the MakerMods vision, building a robot should feel as accessible as snapping together LEGO® bricks, and commanding it should be as simple as writing an English sentence.

In this paper we present a language–to–G-code translation system tailored to off-the-shelf XY gantries and, in the future, MakerMods' ModBlocks ecosystem. We investigate the problem through three neural architectures:

- a **baseline RNN** sequence-to-sequence model,
- a **transformer** encoder–decoder model, and
- a **pre-trained decoder-only** model fine-tuned with Low-Rank Adaptation (LoRA).

Ultimately we hope that this paper represents a step towards natural language interfaces for precise robotic systems. Our findings lay the groundwork for progressively lowering the technical barriers that separate everyday students, hobbyists, small-business and owners from the full power of automated motion platforms.

2 Data

For our model we require entries of data which map natural human instructions to correct and executable G-code corresponding to that instruction. Because this is a relatively unique task there were no data sets that were publicly available for us to use which met our requirements.

As a result we needed to find some way to compile our own data set to use for training and evaluation for the model. We had two main approaches to this problem which we will discuss in this section.

2.1 LLM Generation

Our initial idea was to generate data using available LLM's. In trying to do so we wanted to simplify the output to be a condensed version of G-code that only included the following tokens: `<BOS>`, `<EOS>`, `<ROTATE>`, `<MOVE>`, `<SPEED>`, `0-9`, `<space>`, `<decimal>`, `<EC>`, `<SC>`.

We quickly realized that this was not a feasible approach to generating enough data to train a

[1] https://www.makermods.ai

model in any relevant manner. The LLM's we tried to leverage were simply unable to generate these sequences, otherwise they could already be used to do our task. They had trouble with adhering to the rules of our output grammar at the basic level, and even the "valid" sequences they outputted had significant errors. In general we noticed three major issues with the sample data generated by LLM's:

(a) It failed at generating unique entries when asked for bulk files, often repeating the same 6–10 entries hundreds of thousands of times.

(b) The machine code associated with a command was often completely incorrect for anything beyond the most simple tasks.

(c) It would often give us illegal tool calls that would not adhere to the constraints of the robotic system.

Because of these problems we quickly realized that our original plan for generating data was not going to work. This led us to the approach we will talk about in the next subsection.

2.2 GitHub File Scraping

Instead of trying to generate data in our own defined vocabulary, we looked at a popular programming language called G-code that is used to control gantry-type machines such as 3D printers and CNC machines; G-code typically uses a similar way as our self-defined tokens to control movement, such as `G1` which is akin to the `<MOVE>` command. We looked to find existing files that already had working and correct G-code in them. The data set we used was *The Stack v2* dataset compiled as a part of the Big Code project.

We then generated data entries by web-scraping the GitHub repositories that these G-code files we sourced from and used their file name, repository names, and comments to infer the instructions of the corresponding G-code file. We found that current LLM models were much more capable of parsing the descriptions of these files and rewording them into instructions which the G-code was responsible for.

Below is a simple example of one data entry, although most actual samples are significantly longer than this.

```
{
  "prompt": "3D print a small mechanical
      part.",
  "completion": "G28\nM104 S195\nM140
      S40\nM190 S40\nM109 S195\nG0 F1000
       X0 Y0 Z10\nM104 S0\nM140 S0\nG91\
      nG1 E-1 F300\nG1 Z+0.5 E-5 X-20 Y
      -20 F2700\nG28 X0 Y0\nM84\nG90\nG0
       F9000 X0 Y0 Z0"
}
```

Listing 1: Example data entry from Gantry Instruction Dataset

3 Models

To address our task of translating natural English instructions into executable G-code for an XY Gantry system we employ the use of a few different neural models. In our project we explored three different types of models, all of which are sequence-to-sequence models that map a sequence of natural English instructions to a sequence of tool calls or G-code.

Each of these models was designed with a specific intent in mind and with a specific architecture. All of these decisions and the models themselves are explained in the following sections.

3.1 RNN Baseline

Our initial baseline model was an RNN model that was designed to be trained on our original data entries the shortened output vocabulary. This was a very minimal model that took an input with a dimension of the size of the input vocabulary and an output of the size of the simplified token space.

In between the inputs and outputs was a hidden state made up of 64 RNN cells with about 100 000 parameters. We chose to use a RNN model as they are relatively simple to set up and have an inherent design to processes sequential data making it a good fit for sequence-to-sequence tasks like our translation problem. These models also generally use up less memory and have less parameters than something like a transformer model.

In mathematical terms, we explain our RNN model as follows:

$$\mathbf{h}_0 = \mathbf{0}$$
$$\mathbf{e}_t = \text{one\_hot}(x_t)$$
$$\mathbf{h}_t = \sigma(W_e\mathbf{e}_t + U_h\mathbf{h}_{t-1} + \mathbf{b}_h)$$
$$\mathbf{z}_t = W_o\mathbf{h}_t + \mathbf{b}_o$$
$$Pr(y_t|\mathbf{x}_{1:t}) = \text{softmax}(\mathbf{z}_t)$$

x_t is the input token at time t

e_t is the one-hot vector embedding for the token at time t

h_t is the hidden state up to time t

W_e, U_h are the weight matrices from the input and from the previous hidden state
b_h is some bias value
z_t are the non-normalized logits for the output at time t
$Pr(y_t|x_{1:t})$ is the probability distribution for the output at time t

With this model we made no significant progress towards solving our problem. It was clear that not only was our original dataset far from sufficient, but this model was also not going to get us anywhere due to its lack of complexity and the small parameter space.

After finding a new method for creating data we chose to pivot our focus to solving the problem using our more comprehensive GitHub scraped G-code data. Because of this dramatic shift in the entire problem, for intents and purposes this model will not be compared to the other two as they are simply designed to solve different problems.

3.2 Transformer Model

After the changes we made in the format and collection strategy for data collection we decided to try improving our performance by transitioning to an in-house transformer model rather than a Recurrent Neural Network. Because this is our first model that is designed and trained to act on our new input and output space, it will essentially be treated as our true baseline when being compared to later models.

Our transformer model has 6 hidden layers with dimension of 1024, alongside 4 attention heads, and an embedding space of 256. In general this gives us a model with just under 11 000 000 parameters, a significant increase from our previous RNN.

Once again we can present this model in mathematical terms as follows:

$$\mathbf{e}_t = \text{TokenEmb}(x_t) + \text{PosEmb}(t)$$
$$Q = \mathbf{X}W^Q, K = \mathbf{X}W^K, V = \mathbf{X}W^V$$
$$\text{Attention}(Q, K, V) = \text{softmax}\Big(\frac{QK^\top}{\sqrt{d_k}}\Big)V$$
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$
$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$
$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$
$$\text{LayerNorm}(x + \text{SubLayer}(x))$$
$$\mathbf{H}^{(l)} = \text{EncoderLayer}(\mathbf{H}^{(l-1)})$$
$$\mathbf{Y}^{(l)} = \text{DecoderLayer}(\mathbf{Y}^{(l-1)}, \mathbf{H}_{\text{enc}})$$
$$\mathbf{z}_t = \mathbf{H}_{\text{dec},t}W_o + b_o$$
$$P(y_t|\mathbf{x}, \mathbf{y}_{<t}) = \text{softmax}(\mathbf{z}_t)$$

x_t is the input token at time t
$\mathbf{e}_t$ is the embedding plus positional encoding of x_t
Q, K, V are the query, key, and value matrices derived from input embeddings
W^Q, W^K, W^V are learned projection matrices for each attention component
d_k is the dimensionality of each attention head
head_i is the i^{th} attention head output
W^O is the projection matrix after concatenating all heads
FFN is the position-wise feed-forward network
W_1, W_2 are weight matrices in the FFN, b_1, b_2 are biases
LayerNorm is applied after adding the residual connection
$\mathbf{H}^{(l)}$ is the encoder hidden state at layer l
$\mathbf{Y}^{(l)}$ is the decoder hidden state at layer l
$\mathbf{H}_{\text{enc}}$ is the final encoder output
$\mathbf{H}_{\text{dec},t}$ is the decoder output at time t
W_o, b_o are the weights and bias of the output projection layer
$P(y_t|\mathbf{x}, \mathbf{y}_{<t})$ is the predicted probability of the output token at time t

We trained this model and tracked its average loss per token across many epochs. This is displayed in Figure 1. In doing so we noticed that the model was unable to improve in any significant manner. Although it decreased the loss per token across epochs they were insignificant changes when it came to actual functionality. In fact at the end of training, while inferring with the model it would simple default to returning ' ' as every character, likely because this is the most common character in the data set. We deduced that this was once again likely due to the simplicity of the model also simply the amount of knowledge it has to start.

In solving this task, one of the sub-tasks is to first parse and understand the English instructions from a token level in the first place. When considering this problem it is clear to understand that this task would be computationally unfeasible for this model. At this point, we have a model that serves as a baseline, with no significant use before we transition to our pre-trained model and showing the differences in performance between the two different approaches.

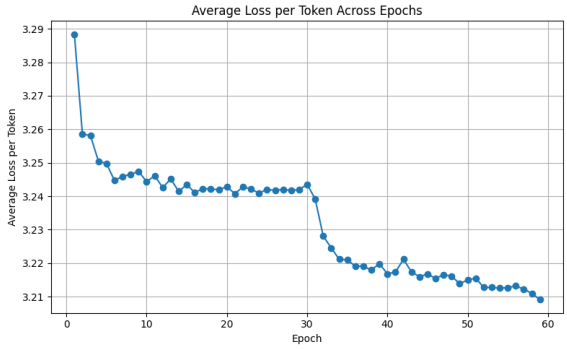


Figure 1: Transformer training progress.

3.3 StarCoder Pretrained Model

StarCoder is a family of large language models for code released by the BIGCODE project. The public checkpoints (3B, 7B and 15B parameters) were trained on more than 80 programming languages, GitHub commits and issues. The largest checkpoint (15B) surpasses models such as LLAMA 65B, CodeGen-16B-Mono and OpenAI's `code-cushman-001` on MBPP and HUMANEVAL benchmarks, while the smallest (3 B) fits on a single consumer GPU.

Zero-shot behavior. We first probed the **StarCoder-3B** checkpoint in a zero-shot setting on a held-out set of 80 natural-language → G-code pairs. Token-level accuracy averaged 0.68, but manual inspection revealed a systematic failure: the model frequently emitted *non-G-code* output (e.g. Python, pseudo-math, or natural language) even when the structure appeared syntactically valid. Examples are shown in Listing 2. This confirmed that task-specific adaptation is required.

```
{
  "prompt": "move to (-94, 52) at 365 mm
      /s",
  "generation": [
    "import matplotlib.pyplot as plt",
    "import numpy as np",
    "# ..."
  ]
},
{
  "prompt": "inside an equilateral
      triangle side 88 mm, draw a circle
       radius 44 mm",
  "generation": [
    "Solution",
    "",
    "The area of the circle is:",
    "$A = \\pi r^2$"
  ]
}
```

Listing 2: Zero-shot StarCoder outputs that are *not* valid G-code.

3.4 Parameter-Efficient Fine-Tuning

The raw StarCoder checkpoint contains 3.7 B trainable weights, requiring ~11.8 GB of VRAM in full precision—beyond the limits of our single RTX 4090. We therefore adopted **parameter-efficient fine-tuning (PEFT)** with Low-Rank Adaptation (LORA) and 4-bit weight quantization (BITSANDBYTES).

We curated 2,200 *instruction, G-code* pairs scraped from open-source GitHub repositories in THE STACK V2. Every sample was manually or heuristically filtered to ensure syntactic validity and to remove machine-specific artifacts (e.g. tool change commands irrelevant to an XY gantry).

LoRA rank $r = 8$ was used for all self-attention and feed-forward projection matrices, yielding 227 M trainable parameters (∼6.1% of the full model). Training hyper-parameters are summarized in Table 1.

Batch size (micro)	4
Learning rate	2e-5
Scheduler	cosine w/ 500 warm-up steps
Max sequence length	2048 tokens
Quantization	4-bit NF4
Total update steps	8 500 (≈15 epochs)
Wall-clock time	3 h on one RTX 4090

Table 1: Fine-tuning hyper-parameters.

4 Results

4.1 Quantitative Evaluation

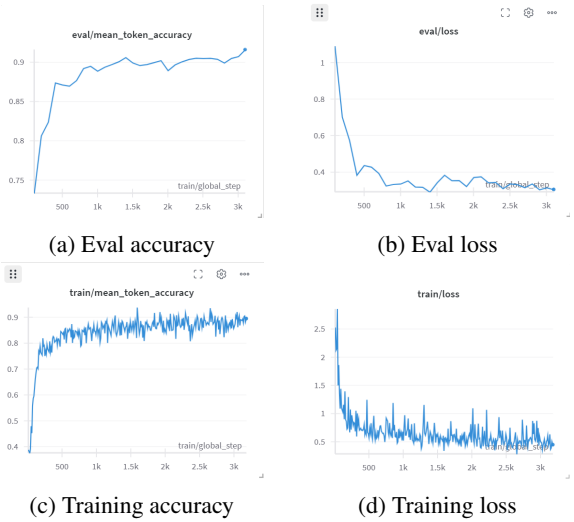
Fine-tuning increased mean token accuracy on the held-out evaluation set from **0.68** → **0.92**. Average loss per token fell to **0.28**. The training and evaluation curves are shown in Figure 3.

4.2 Qualitative Examples

Below we present four representative generations from the fine-tuned model. Each listing is followed by a reserved space where the corresponding toolpath visualization can be inserted.

```
; draw a circle with the radius of 54 mm
G2 I54  J0  F1923
G2 I-54 J0  F1923
```

Listing 3: Example 1 — draw a circle of radius 54 mm.



(a) Eval accuracy (b) Eval loss

(c) Training accuracy (d) Training loss

Figure 2: Evolution of both accuracy and loss across 3,000 epochs for the train and test sets.

Figure 3: Mean-token accuracy and evaluation loss during fine-tuning.

```
; move to (-94,52) at 365 mm/s
G1 X-94 Y52 F365
```

Listing 4: Example 2 — move to `(-94, 52)` at 365 mm/s.

```
; inside a equilateral triangle side 88
    mm, draw a circle radius 44 mm
G1 X88 Y0  F1317
G1 X44 Y74 F1317
G1 X0  Y0  F1317
G2 I44  J0  F1273
G2 I-44 J0  F1273
```

Listing 5: Example 3 — inside an 88 mm equilateral triangle, draw a circle of radius 44 mm.

```
; draw a equilateral triangle side 45 mm
G1 X45 Y0  F923
G1 X22 Y39 F923
G1 X0  Y0  F923
```

Listing 6: Example 4 — draw an equilateral triangle with 45 mm sides.

Error Analysis

Residual errors (token-level accuracy plateaued at ∼92%) are dominated by:

(a) **Limited abstraction ability:** the model copes well with basic primitives (`G0`/`G1` lines and `G2`/`G3` arcs) but fails on higher-level or vaguely specified shapes such as “tree” or “umbrella,” reverting to irrelevant or empty output.

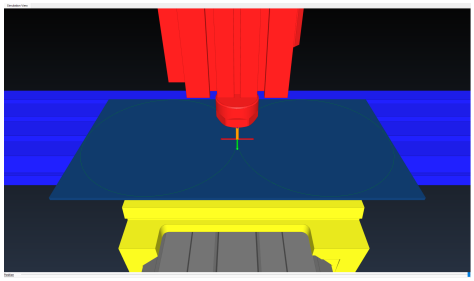


Figure 4: Tool-path visualization for Example 1 (circle, 54 mm).

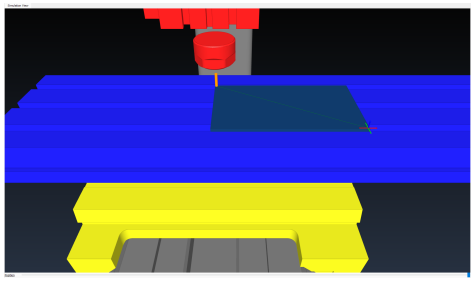


Figure 5: Tool-path visualization for Example 2 (rapid move).

(b) **Primitive duplication:** when asked to *“draw a circle of radius 54 mm”* the model emitted two consecutive `G2` commands, producing two concentric circles instead of the single intended path.

(c) **Spatial-constraint violations:** for *“draw an equilateral triangle **inside** the circle”* the generated code placed the triangle on the build-plate origin and repeated the circle twice, ignoring the containment requirement.

Mitigation strategies include augmenting the training set with composite-shape demonstrations, prompting it with more specific instructions and even coordinates, and using a post-generation syntax and semantic checks.

4.3 Future Work

The current study confirms that a 3B-parameter backbone plus LoRA can map everyday language to valid G-code. We foresee four complementary directions for improvement:

- **Larger backbone & richer data.** Scaling to StarCoder-7B or -15B—with higher LoRA rank and 8-bit (or full-precision) adapters—should increase capacity for complex, abstract shapes (*e.g.,* “tree,” “umbrella”). In parallel we will expand training from the present 2 200 examples to the ∼56 000 publicly available G-code files, greatly enriching

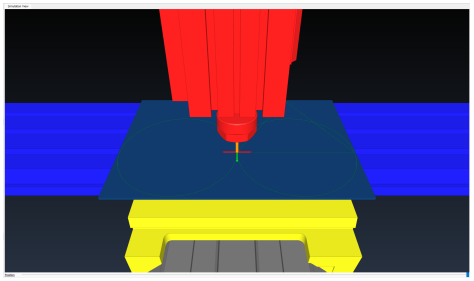


Figure 6: Tool-path visualization for Example 3 (triangle with inscribed circle).

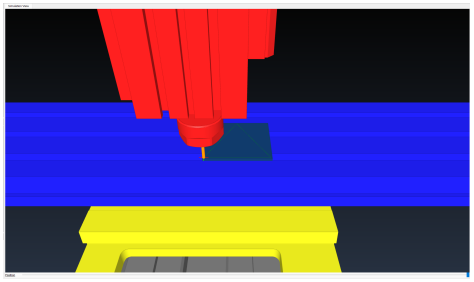


Figure 7: Tool-path visualization for Example 4 (triangle, 45 mm).

geometric diversity.

- **Self-correction loop.** Every generation passes through fast syntax and semantic validators; failures trigger a another iteration of StarCoder call that receives both the original prompt and a concise error description, enabling the model to be self-correcting and "Agentic".
- **Retrieval-Augmented Generation (RAG).** Machine-specific metadata (axis limits, steps-per-mm, tool-changer IDs) will be stored in a vector database. At inference the model retrieves the configuration of the active CNC or 3-D printer, tailoring code to heterogeneous hardware.
- **Path simulation & collision screening.** A lightweight preview engine will simulate the tool-path and flag violations such as out-of-bounds motion or collisions with existing prints; detected problems are fed back into the self-correction loop for automatic repair.

5 Conclusion

We presented an end-to-end pipeline that converts plain-English instructions into executable G-code for XY gantry systems. With parameter-efficient LoRA tuning, a compact 3 B StarCoder backbone reaches **92 %** mean-token accuracy on a single consumer GPU. Our experiments confirm that transformer-based models can grasp both the semantics of natural commands and the syntax of G-code; however, their performance worsens on abstract shapes and machine-specific instructions. These limitations point to clear next steps: larger backbones, richer training corpora, retrieval-augmented prompts, and closed-loop self-correction. Pursuing these directions will transform our prototype into a robust platform for natural-language G-code control.