WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

A LabVIEW–Arduino Closed-Loop Two-Axis CNC Plotter with VISA-Free Serial Communication on Linux

Pranav Dhole, Pranav Rao and Saravanakumar R.

Vellore Institute of Technology, Vellore, India

ABSTRACT.

Closed-loop CNC systems usually pair a real-time motion controller with a supervisory graphical interface. This paper describes a two-axis CNC plotter built on that split. LabVIEW provides the operator interface, target entry, start/stop control and position logging, while an Arduino Uno performs real-time stepper control and position correction using VL53L0X time-of-flight distance sensors. The host runs Arch Linux, which is not a supported platform for the NI-VISA runtime on which LabVIEW's standard serial VIs depend. We describe the problem this causes, explain why installing NI-VISA on an unsupported distribution was not practical here, and present a VISA-free alternative: a small Python bridge exposes the Arduino's serial port on a local TCP socket, so LabVIEW can reach the controller through its native TCP/IP functions. We also present the line-oriented command protocol and the closed-loop control scheme, and we say plainly what is implemented and what still awaits calibration and testing. The bridge has been verified end to end with a minimal test. In preliminary measurements, closed-loop correction halved the mean absolute position error on the X axis (1.0 mm to 0.5 mm) but did not change it on the Y axis (0.8 mm in both modes).

Keywords: LabVIEW; Arduino; NI-VISA; serial communication; Linux; closed-loop control; CNC plotter; time-of-flight sensor

1. Introduction

Low-cost and educational CNC platforms usually divide the work between a host computer and an embedded controller. The host supplies the operator interface and the data logging. The embedded board takes the time-critical parts, namely step timing and sensor sampling. The reason is simple: a general-purpose desktop operating system cannot guarantee the timing that direct stepper drive demands. Splitting the work also makes the firmware reusable, since the same code can sit behind anything from a terminal session to a full graphical application.

LabVIEW is a common choice for that front end in teaching and research laboratories. Its graphical dataflow language and front-panel model make it quick to put together operator interfaces, indicators and plots. The catch is that LabVIEW's standard serial functions sit on top of the NI-VISA runtime, and NI distributes that runtime for only a handful of Linux distributions [1].

This paper reports a two-axis plotter that uses a LabVIEW front end and an Arduino Uno controller on an Arch Linux host. Arch is not among the distributions NI supports for NI-VISA [1], so the usual LabVIEW serial route was closed to us. We explain how we worked around this and then present the system architecture, the communication protocol and the closed-loop control design.

We close the loop because an open-loop stepper drive assumes that every commanded step produces exactly one step of motion. On a low-cost mechanism built from salvaged DVD-drive sleds, friction, backlash and varying load can break that assumption, and missed or extra steps then accumulate silently as position error. Measuring the carriage position directly lets the controller see these discrepancies and correct them instead of integrating them.

Our contributions are modest and practical. First, we document the NI-VISA gap on an unsupported Linux distribution and the options for working around it. Second, we describe a VISA-free serial-to-TCP path that LabVIEW can use through its native TCP/IP functions. Third, we define a small line-oriented command protocol and a closed-loop control structure that keeps all real-time behaviour on the Arduino. Finally, we state plainly which parts of the system are implemented and which still need calibration and testing. Section 2 covers the architecture, Section 3 the compatibility problem, Section 4 the bridge, Section 5 the protocol, Section 6 the hardware setup, Section 7 the closed-loop control, Section 8 the results, and Section 9 concludes.

2 System architecture

2.1 Division of responsibilities

The system has two cooperating parts (Fig. 1). LabVIEW provides the graphical interface. It accepts target X/Y coordinates and start/stop commands from the operator, shows the actual position and the current position error, and logs and plots the position history. The Arduino Uno [2] handles everything

that has to happen in real time: generating step and direction sequences for the stepper motors, sampling the position sensors, computing closed-loop corrections and reporting status back to LabVIEW.

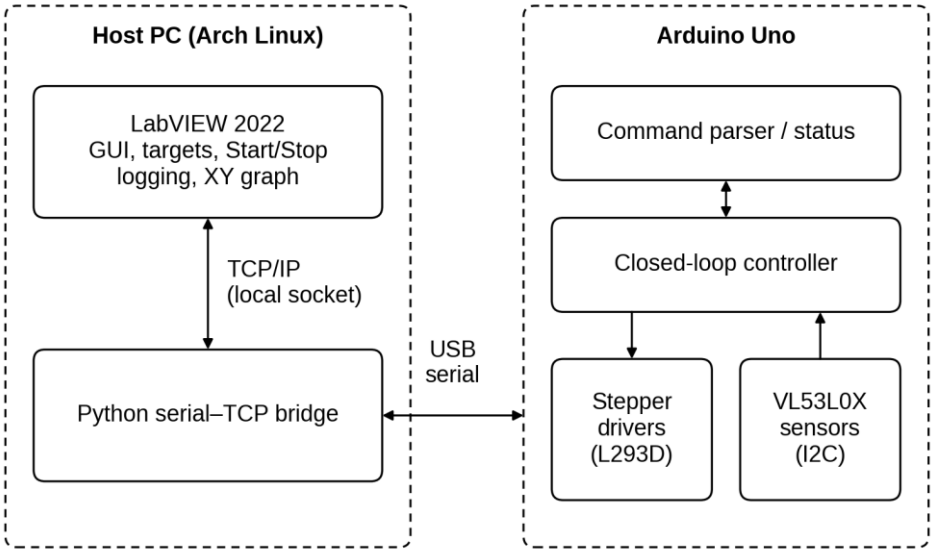


Fig. 1. Block diagram of the plotter: LabVIEW and the Python bridge on the Arch Linux host, and the Arduino Uno controller with its stepper drivers and distance sensors.

LabVIEW never generates individual step pulses. It sends high-level target positions and receives high-level status reports. That keeps the interface between the two halves narrow and low-rate, so scheduling jitter on the host cannot disturb motor sequencing, and it means the front end could be swapped out without touching the firmware.

2.2 Hardware

Each axis is driven by a small stepper motor salvaged from a DVD-drive sled and connected through an L293D-based motor shield [3]. Position feedback comes from one VL53L0X time-of-flight distance sensor [4] per axis. Each sensor is mounted to measure the distance between the moving carriage and a fixed reference surface at one end of that axis's travel, and we take this distance as the carriage's absolute position. In principle this removes the need for a homing routine, because position is measured rather than counted. The caveat is the sensor offset calibration discussed in Section 7. The components are listed in Table 1.

Table 1. Main components and their roles.

Component	Qty	Role
Arduino Uno	1	Real-time controller; USB serial link to the host
L293D-based motor shield	1	Drives both stepper motors
DVD-sled stepper motor	2	One per axis (X and Y)
VL53L0X distance sensor	2	Absolute carriage position, one per axis
Host PC (Arch Linux)	1	LabVIEW 2022 (64-bit) and the Python bridge

Both sensors share the Arduino's single I^2C bus and power up at the same default address (0x29). To tell them apart, each sensor's XSHUT pin is wired to its own digital pin. At power-up both sensors are held in hardware standby through XSHUT. The firmware releases one sensor, assigns it a new address, and then releases the other, which leaves the two devices individually addressable on the shared bus [4]. The assigned address is not retained across power cycles, so this procedure runs at every boot.

The sensor also shapes the mechanical layout. It cannot range reliably at very short distances [4], so the reference surface has to be placed such that the carriage stays within the sensor's reliable working range over the full travel of the axis. Its output is also quantised in millimetres, which matters for the tolerance discussion in Section 6.3.

2.3 Firmware

The Arduino firmware is built on the AFMotor stepper library [3] and a VL53L0X driver library. It parses incoming commands, drives each axis toward its commanded target, and periodically re-samples both distance sensors to correct any discrepancy between commanded and actual position. Three jobs therefore share one processor: command parsing on the serial port, step generation through the motor shield, and distance sampling over I^2C. A distance conversion takes a finite time, typically tens of milliseconds at the sensor's default timing budget [4], and stepping is carried out in short bursts. The firmware consequently alternates between the three jobs instead of running them concurrently. Section 6 describes the resulting sequence.

2.4 LabVIEW application

The front panel provides numeric controls for the target X and Y positions, Start and Stop buttons, indicators for the actual position and the error, a status/error string, and an XY graph fed by logged position samples (Fig. 2). The block diagram talks to the controller only through LabVIEW's native TCP functions (TCP Open Connection, TCP Write, TCP Read and TCP Close [5]) and converts between the front-panel numerics and the line-oriented protocol of Section 5 using string formatting and scanning functions.

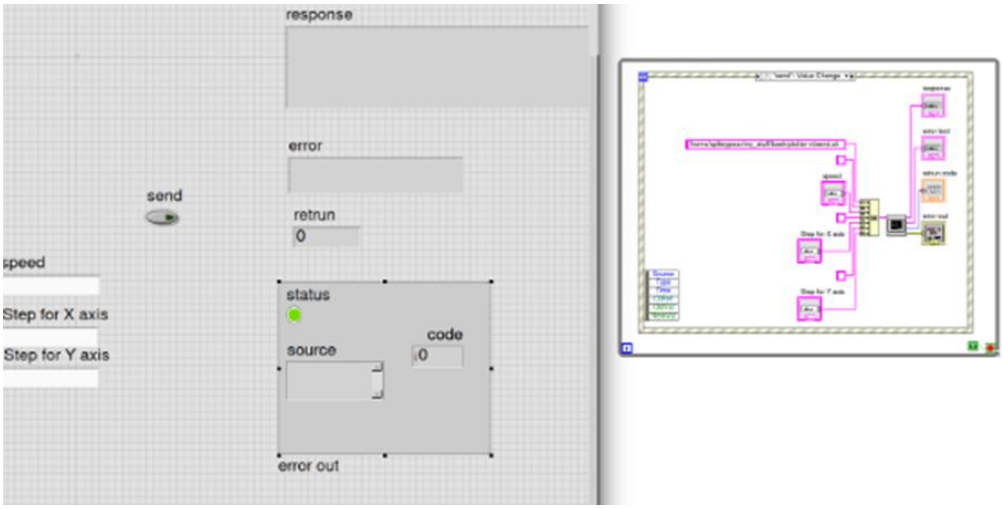


Fig. 2. Schematic layout of the LabVIEW front panel: target controls, Start/Stop buttons, position and error indicators, a status string.

3 The NI-VISA compatibility problem

3.1 Dependence of the serial VIs on NI-VISA

LabVIEW's standard serial VIs (VISA Configure Serial Port, VISA Write, VISA Read and VISA Close) are built on the NI-VISA runtime, which is installed separately from LabVIEW. On Linux, NI provides NI-VISA for a small set of distributions: Red Hat Enterprise Linux, openSUSE Leap and Ubuntu LTS releases. It is installed through distribution-specific package repositories, with kernel modules built against the running kernel through DKMS [1]. Arch Linux is not on that list, and the official Arch repositories contain no NI-VISA package.

The Arch User Repository (AUR) does carry a community-maintained ni-visa package [8], but NI does not support it. The PyVISA documentation calls it an early-development package known to work for USB and TCP/IP interfaces and does not mention serial ports [9], and a note on the package page says that it does not install NI's kernel modules [8].

3.2 Observed failure

On the target machine we installed LabVIEW 2022 (64-bit), and its bundled VISA example VIs were present. No VISA shared library could be found under the LabVIEW installation tree or in the system library paths. Running a basic VISA test VI returned error −1073807202 (hexadecimal 0xBFFF009E, VI_ERROR_LIBRARY_NFOUND), which means that a library required by VISA could not be located or loaded [10]. The VISA VIs are therefore present but unusable. The LabVIEW installation itself is intact; only the separately distributed runtime is missing.

3.3 Options considered

We looked at three routes to a working serial link (Table 2). The first was to adapt NI-VISA to Arch Linux, which we judged impractical. NI's DKMS-based packages assume the kernel-header and packaging conventions of a supported distribution, and the only community package we know of is neither supported by NI nor documented for serial use (Section 3.1). Adapting NI's packages to a rolling-release distribution would also turn a one-time installation into continuing maintenance, which is out of proportion for a project of this size. The second route was to host LabVIEW on a supported

distribution. That would satisfy NI-VISA but gives up the Arch Linux host that motivated the work in the first place. The third was to replace the VISA serial layer with a TCP link to a small relay process. This removes the dependency altogether, and it is the route we took.

Table 2. Routes considered for a LabVIEW serial link on Arch Linux.

Option	Requirement	Outcome
Port NI-VISA to Arch Linux	Adapt NI's DKMS packages to Arch kernel-header and packaging conventions	Judged impractical
Move LabVIEW to a supported distribution	Red Hat, openSUSE Leap or Ubuntu LTS host [1]	Not adopted
Serial–TCP bridge with native TCP VIs	One Python process on the host	Adopted

Two other VISA-free routes deserve a mention. LabVIEW's Call Library Function Node [12] can call the C library's serial (termios) routines directly, which avoids the extra process but moves low-level port handling into the block diagram. A general-purpose relay such as socat [11] can expose a serial device on a TCP port with a single command line and no custom code, so a hand-written bridge is not strictly necessary for the basic relay. A Python implementation remains convenient here: it keeps the relay logic short and readable, it lets us handle the Uno's reset-on-open behaviour explicitly (Section 4.2), and it gives us an easy place to add timestamping for the latency measurements planned in Section 7.

4 VISA-free communication

4.1 Bridge architecture

Instead of pursuing NI-VISA on an unsupported platform, we carry the communication between LabVIEW and the Arduino over a local TCP socket, using LabVIEW's native TCP/IP VIs, which have no dependency on VISA. A lightweight Python process on the same host opens the Arduino's serial device node and relays bytes between the serial connection and a TCP socket to which LabVIEW connects. This bridge is the only component that touches the serial device directly. From LabVIEW's point of view, the Arduino is simply a TCP peer. The implementation relies on the pySerial package [6] for serial access and Python's standard socket module [7] for the TCP endpoint.

4.2 Design considerations

The bridge is transparent by design. It forwards bytes in both directions without interpreting them, so the protocol of Section 5 lives entirely in LabVIEW and the firmware and can change without any modification to the bridge. The two directions are relayed independently, so status lines reach LabVIEW as soon as the Arduino emits them, not only in reply to a request.

Three practical points follow from the hardware and the platform. First, an Arduino Uno normally resets when its serial port is opened. The bridge should therefore hold the port open for the whole session and give the firmware time to boot before relaying traffic, because reopening the port for every command would restart the firmware each time. Second, the account running the bridge needs permission to open the serial device node. On Arch Linux that normally means membership of the group that owns it, which is uucp. Third, since the bridge is meant only for the local machine, its socket should be bound to the loopback interface so that the plotter is not exposed to the network.

4.3 Validation

We validated the approach with a minimal test protocol before implementing the full command set. LabVIEW opens a TCP connection to the bridge, writes the character '1', and reads back the string "HELLO" sent by the Arduino. This confirms the full path from the LabVIEW front panel through the bridge and the serial link to the firmware and back (Fig. 3a). The test exercises every link in the chain (TCP connect, TCP write, serial write, firmware response, serial read and TCP read) while needing no protocol parsing, so any failure could be attributed to the transport rather than to application logic.

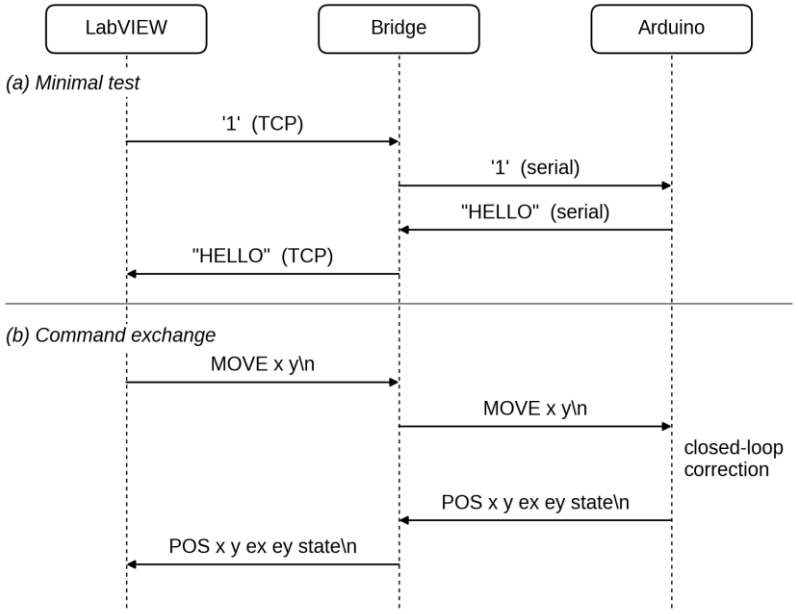


Fig. 3. Message sequence between LabVIEW, the bridge and the Arduino: (a) the minimal test used to validate the bridge; (b) a command exchange using the protocol of Section 5.

5 Communication protocol

On top of the same TCP link, the plotter uses a small fixed-format ASCII protocol. Messages are newline-terminated so that LabVIEW can parse each line with simple string-matching and scanning functions. Table 3 lists the messages in each direction, and Fig. 3b shows a typical exchange.

Table 3. Communication protocol between LabVIEW and the Arduino controller.

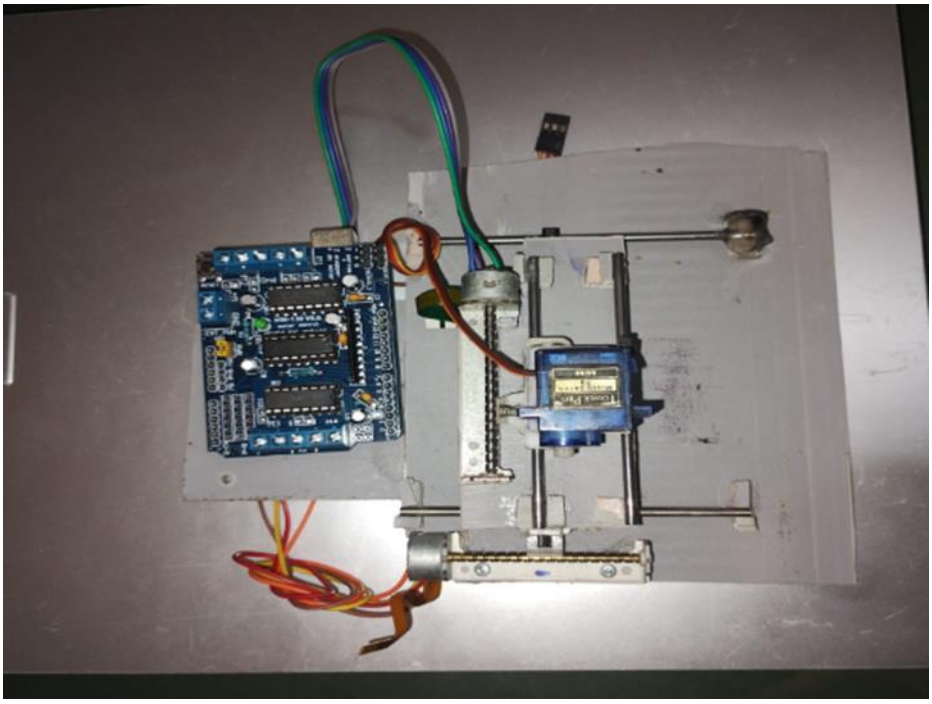
Direction	Message	Meaning
LV → Ard	MOVE x y\n	New target position, in millimetres
LV → Ard	STOP\n	Halt motion and release the motors
LV → Ard	START\n	Resume or enable motion
LV → Ard	STATUS?\n	Request one status line
Ard → LV	POS x y ex ey state\n	Actual X, actual Y, error X, error Y, controller state
Ard → LV	ERR code msg\n	Fault report

Every message is a single line of printable ASCII, with fields separated by single spaces. Positions and errors are given in millimetres, so the operator works in physical units and the steps-per-millimetre conversion stays inside the firmware. A MOVE command replaces the current target. STOP halts motion and releases the motors (Section 6.4), START resumes or enables motion, and STATUS? asks the controller for one POS line. A POS message carries the actual X and Y positions, the corresponding errors and a controller-state field, which is all that the front-panel indicators in Fig. 2 need. An ERR message carries a fault code and a short text that LabVIEW can route to the status/error indicator.

Keeping the protocol human-readable also pays off during bring-up: any serial terminal, or any TCP client connected to the bridge, can exercise the firmware by hand without the LabVIEW application. And because the protocol is defined purely in terms of text lines, it does not depend on the transport beneath it. The same messages would work unchanged over a direct serial connection on a platform where NI-VISA is available.

6. Hardware setup

We assemble the plotter frame with two salvaged DVD-drive stepper motors, mount a VL53L0X ToF sensor on each of the X and Y axes, and connect the Arduino Uno to the L293D motor shield. Both sensors start with the same I2C address, so the shutdown (XSHUT) pin of each sensor is wired to its own Arduino pin, which lets the sensors be started one at a time and given different addresses.



7. Closed-loop position control

After receiving a MOVE command, the firmware enters a control loop (Fig. 4). On each pass it samples both VL53L0X sensors, computes the difference between the commanded target and the measured position on each axis, and issues a bounded number of motor steps toward the target. The loop runs until both axes are within a configured position tolerance, at which point the controller reports an idle state. A STOP command halts motion and releases both motors.

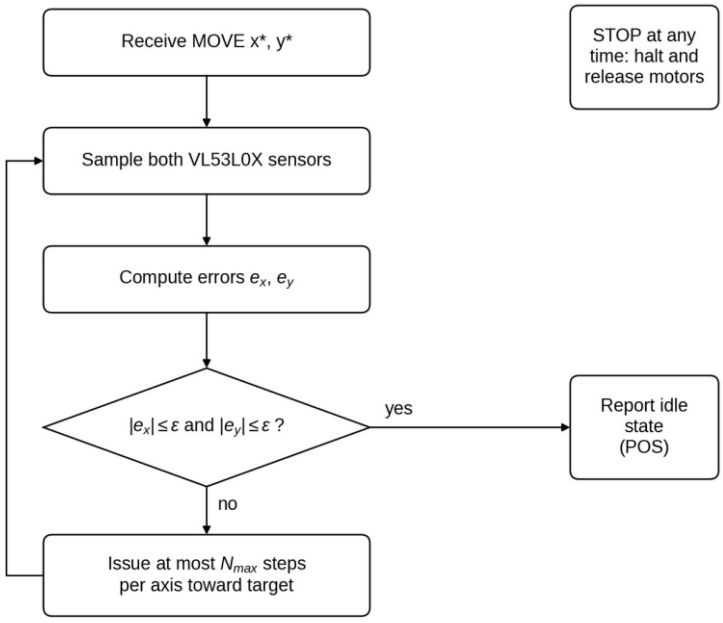


Fig. 4. Flow chart of the closed-loop position control performed by the Arduino firmware after a MOVE command.

7.1 Position measurement

For axis $i \in \{X, Y\}$, the measured position at pass k is obtained from the raw sensor distance d_i and a per-axis offset δ_i that relates the sensor's zero to the origin of the plotter coordinate system:

$$x_i(k) = d_i(k) - \delta_i \qquad (1)$$

Until the offset calibration of Section 7 is done, δ_i is a placeholder, and the reported positions are referenced to the sensor rather than to a calibrated origin. Equation (1) assumes that the measured distance grows with the coordinate. Where a sensor is mounted the other way round, the relation becomes $x_i = \delta_i - d_i$. In both cases the stepping direction has to be chosen so that a positive step count increases x_i.

7.2 Correction law

With x_i^* denoting the commanded target, the error on each axis is

$$e_i(k) = x_i^* - x_i(k) \qquad (2)$$

and the bounded correction issued on that axis in pass k is

$$n_i(k) = \operatorname{sgn}(e_i(k)) \cdot \min\{ N_{\max}, \lceil \kappa_i \, |e_i(k)| \rceil \} \qquad (3)$$

where κ_i is the steps-per-millimetre conversion used by the firmware for axis i, $N_{\max}$ is the maximum number of steps issued per axis in one pass, and the sign of n_i selects the stepping direction. Equation (3) is the simplest form consistent with the bounded-step behaviour: a step count proportional to the error, saturated at $N_{\max}$. The loop ends when both axes satisfy

$$|e_X(k)| \le \varepsilon, \qquad |e_Y(k)| \le \varepsilon \qquad (4)$$

where ε is the configured position tolerance. The controller then reports its idle state in a POS message.

7.3 Tolerance and stability

Two effects set a lower bound on the tolerance ε. The first is the resolution of the stepping mechanism: one step moves the carriage by $1/\kappa_i$, so a tolerance below that cannot in general be met. The second is the noise and quantisation of the time-of-flight sensor, because the loop acts on the measured position rather than the true one. Common driver libraries return the range as a whole number of millimetres, so a tolerance much below 1 mm is unlikely to be meaningful whatever the step size. If ε is set below either limit, the controller can keep issuing corrections of alternating sign around the target and never reach the idle state. Backlash in the DVD-sled mechanism has a similar effect, because a reversal of direction consumes steps before the carriage starts to move.

The step limit $N_{\max}$ matters for a different reason. Consider an unsaturated pass and ignore rounding, sensor noise and backlash. If one commanded step actually moves the carriage by $1/\kappa_i^{\text{true}}$, the error evolves as

$$e_i(k+1) = (1 - g_i)\, e_i(k) \qquad (5)$$

with $g_i = \kappa_i/\kappa_i^{\text{true}}$, the ratio of the steps-per-millimetre value in the firmware to the true value of the mechanism. To first order the error therefore decays without changing sign if $0 < g_i \le 1$, decays with alternating sign if $1 < g_i < 2$, and fails to converge if $g_i \ge 2$. A well-calibrated κ_i gives g_i close to 1. When the calibration is poor, an uncapped correction would overshoot by a distance proportional to the whole error. The step limit $N_{\max}$ caps that overshoot at a fixed number of steps, which is why it is applied even when the error is large. The price of a small $N_{\max}$ is that long moves need more passes, each including a sensor reading, and so take longer.

7.4 Stop behaviour

A STOP command is handled outside the correction law. It halts motion and releases both motors, so that the carriage can be moved by hand and the coils are not left energised. Each correction pass is a short burst of steps followed by a sensor reading, and the firmware checks the serial port between passes. A STOP therefore takes effect at the next check, which at worst comes after the current burst and sensor reading. Because the passes are short, this delay should be small, although we have not yet measured it.

8. Results and discussion

8.1 Comparison with the conventional approach

This section reports what the current prototype has demonstrated and compares the design with the conventional alternative: an open-loop stepper drive behind a VISA-based serial link. Table 4 summarises the comparison. The status column separates results verified on the hardware from features that are implemented but not yet quantified.

Table 4. Comparison of the proposed system with a conventional open-loop, VISA-based design.

Aspect	Conventional approach	This system	Status
LabVIEW serial access on Arch Linux	VISA VIs fail with error −1073807202 (library not found)	Native TCP VIs through a Python serial–TCP bridge; no VISA library needed	Verified (HELLO test)
Host platform	Move LabVIEW to a distribution NI supports	Arch Linux host retained	Verified
Position feedback	None; commanded steps are assumed to equal motion	One VL53L0X distance sensor per axis measures the carriage position	Implemented
Missed or extra steps	Accumulate silently as position error	Measured and corrected on every pass until both errors are within ε	Implemented; X error halved, Y unchanged (Table 5)
Homing	Required to establish an origin	Not required in principle, because position is measured	Subject to offset calibration
Two identical sensors on one I²C bus	Address conflict at 0x29	XSHUT sequencing assigns distinct addresses at every boot	Implemented
Real-time behaviour	Depends on the host	All step timing and sensor sampling on the Arduino	By design

The one result verified end to end is the transport. LabVIEW opens a TCP connection to the bridge, writes the character 1 and reads back HELLO from the Arduino (Section 4.3). With this path in place the LabVIEW application calls no VISA VI, so the library-not-found error no longer arises on the target machine and the Arch Linux host is kept.

8.2 Problems revisited

Three problems motivated this work. The first was that NI-VISA is not available for Arch Linux, which left LabVIEW without a serial link. The serial–TCP bridge removes the dependency, and the minimal test confirms that the path works. The second was that an open-loop stepper drive on a salvaged DVD-sled mechanism cannot detect missed or extra steps. The closed-loop controller of Section 7 measures the carriage position directly and corrects the error on each pass, so such errors are corrected instead of accumulated. The stability analysis of Section 7.3 shows that the error decays whenever the firmware's steps-per-millimetre value stays within a factor of two of the true value ($g < 2$ in Eq. 5), whereas an open-loop drive has no corrective mechanism at all. Section 8.3 reports preliminary measurements of how much accuracy it buys. The third problem was that two identical sensors share one I²C bus, and the XSHUT start-up sequence of Section 2.2 resolves it.

Not every limitation is closed. The sensor offsets δ_i still need calibration, travel-limit protection is not yet in place, and the 1 mm quantisation of the sensor bounds the tolerance ε that can be met (Section 7.3).

8.3 Quantitative evaluation

Table 5 reports preliminary measurements of the mean absolute position error with the closed-loop correction disabled (open loop) and enabled.

Table 5. Mean absolute position error with and without closed-loop correction (preliminary measurements).

Metric	Open loop	Closed loop	Reduction
Mean absolute position error, X axis (mm)	1.0	0.5	50%
Mean absolute position error, Y axis (mm)	0.8	0.8	0%
Mean of the two axes (mm)	0.90	0.65	28%

On the X axis, closed-loop correction halved the mean absolute error, from 1.0 mm to 0.5 mm. On the Y axis the mean error was 0.8 mm in both modes, so no improvement was observed there. Averaged over the two axes, the mean error fell from 0.90 mm to 0.65 mm, a reduction of about 28%. The largest single error observed was 2 mm .

We did not investigate why the Y axis showed no improvement. Two explanations follow from Section 7.3 and remain hypotheses. First, a mean error of 0.8 mm is already below the 1 mm quantisation of the sensor reading, so the loop may have little left to correct. Second, backlash or a poorly calibrated steps-per-millimetre value on that axis may offset the benefit of feedback. Repeatability and time to reach the target were not measured, and the bridge latency and STOP delay (Section 7.4) are also pending. These measurements are left for future work.

9 Conclusion

We have described how a LabVIEW-based CNC controller can be built on a Linux distribution that NI does not support for NI-VISA, by removing the VISA dependency altogether. A minimal serial-to-TCP bridge, combined with LabVIEW's native TCP/IP VIs, reproduces the point-to-point communication that the VISA serial VIs would otherwise provide, while all time-critical motor control and sensor sampling stays on the Arduino. The transport has been verified end to end with a minimal test.

The price of the approach is one additional process on the host, and the cost of that extra hop has not yet been measured. In return, the LabVIEW application depends only on functions that are available on every platform LabVIEW supports, and the same pattern should carry over to any serial device that LabVIEW needs to reach from a Linux distribution without NI-VISA support. Preliminary measurements show that closed-loop correction halved the mean X-axis error but left the Y-axis error unchanged. Calibration, travel-limit protection, a larger and repeatable test set, and an investigation of the Y-axis result are the next steps.

References

[1] National Instruments, "NI-VISA Linux operating system support," NI product documentation.

[2] Arduino, "Arduino Uno Rev3," technical specifications.

[3] Adafruit Industries, "Adafruit Motor/Stepper/Servo Shield for Arduino v1," product documentation.

[4] STMicroelectronics, "VL53L0X: Time-of-Flight ranging sensor," datasheet.

[5] National Instruments, "LabVIEW TCP and UDP functions," LabVIEW product documentation.

[6] C. Liechti, "pySerial documentation."

[7] Python Software Foundation, "socket: Low-level networking interface," Python documentation.

[8] "ni-visa," Arch User Repository (AUR) package page. [Online]. Available: https://aur.archlinux.org/packages/ni-visa

[9] PyVISA developers, "NI-VISA installation," PyVISA documentation (FAQ). [Online]. Available: https://pyvisa.readthedocs.io/en/latest/faq/getting_nivisa.html

[10] Keysight Technologies, "VISA error codes," IO Libraries Suite documentation. [Online]. Available: https://helpfiles.keysight.com/IO_Libraries_Suite/English/IOLS_Windows/VISA/Content/visa/VISA_Error_Codes.htm

[11] G. Rieger, "socat: Multipurpose relay," manual page.

[12] National Instruments, "Call Library Function Node," LabVIEW product documentation.