LabView Notes

Notes on LabView applications and integrations thereof. Hopefully using NI equipment won't be terribly difficult.

Artur R. B. Boyago

2024-11-27

Preface

Contents

Part 1: How LabView operates

LabView is a graphical development environment, where one draws diagrams representing certain state machines. Programs in LV are are directed functional graphs containing certain pre-packaged functionalities1; these are all represented inb *.vi* (Virtual Instrument) files, where each *VI* represents a simple subroutine block. The content of the represented code is both the graphical (diagrammatic) representation, and the eventual compiled information (inplaceness information). One can also purposely separate the production of the two starting from LV2010.

The compilation pipeline for *G (dataflow)* code (not to be confused with CNC G-code), the language of the actual drawn diagrams, involves a *type propagation alogithm*. All diagrams are converted to a high-level FDIR graph languaage. This "sequent graph calculus" is useful since its more amenable to particular machine-instruction rleated refinement steps not possible in G code. The so-called "*DFIR decompositions*" would correspond to intuitive logical steps.

The type propagation checks the validity and optimizes the graph, as well as to then generate an intermediate sequent calculus representation to be handled by a LLVM compiler, producing machine code directly into an ´.exe´.

1.1. Actor Frameworks

Part 2: Interoperability with different languages

The basic way to operate on G NV code is by means of external DLLs that are compatible with C, which is naturally interoperable wiyth G. We'll consider two very different languages, *Rust* and *Python*, altough this is possible on virtually any language, including *OCaml* and *Haskell*.

Rust is famous for its very good performance and intrinsic memory safety, as well as having many pre-packaged functionalities and existing libraries. Python, being a dynamically typed scripting language, is far slower but easier to make more sophisticated programs.

2.1. Rust

For Rust, we need to produce a C-compatible DLL. This can be done by explictly directing the compiled code into a `[lib]` package in the `cargo.toml` file of any Rust project [1].

```toml
[package]
name = "demo_dll"
version = "0.1.0"
edition = "2021"

[dependencies]

[lib] // Library target settings
name = "demo_dll"
rate-type = ["cdylib"] // Library for C/C++ integration
```

Listing 2.1: Basic _msnifest_ for a standard C compiled DLL.

By specifying `[cdylib]` we produce the necessary C ABI as a dynamic library, DLL. In order to expose this functionality to other softwares, like LabView, all the relevant functions mut be *external functions* [2]. One can further specific dynamic runtime architecture, such as with

`[arm-unknown-linux-musleabi]` and the like, altough LV automatically does so in its internal compilation process.

When actually wrtiting down functions for LabView, use the `#[no_mangle]` attreibute as well to facilitate linking, as it forces the produced libraries to retain the precise function names for calling.

```rust
#[no_mangle] // Don't obfuscate function names

pub extern "C" fn add_numbers(a: i32, b: i32) -> i32 {
    return a + b;
}
```

Listing 2.2: A simple external function that adds and returns a `i32`.

When in LV, use the ***Call Library Function Node*** and summon the produced DLL. Carefully add the parameters with the proper typing in the *parameters* menu; for example, if you selected a `i32` return type, guarantee you're using a 32-bit signed integer.

2.2. Python

LabView naturally comes with Python support; it contains three nodes, an *Open/Close Python Session* and *Python Call* node. When placed sequentially, one specifies the Python version and filepath for the programs in question, and then one can call the functions quite easily.

Part 3: Motor controls

Suppose we have n points $\vec{\boldsymbol{x}} \in \mathbb{R}^m$ arranged as $[\boldsymbol{A}]_{\{mn\}}$ and we want to *interpolate* a curve $\gamma : [a, b] \to \mathbb{R}^m$ through them. We may perform any of the number of different parametric interpolations such as *linear, polynomial, non-linear* and the likes.

3.1. GCODE controls

For the *machine + laser* control, we'll need a simpler subset of linear motion, and reinterpret certain parameters for the laser shutter. It is convenient to deploy some methodology to smoothly modulate the shutter and other equipment as well.

The standard list is of the following X commands:

- General configurations
  - G21 = Milimeters
  - G28 = Return Home
  - G90 = Absolute Mode
  - G91 = Relative Mode (Deired for Laser applications)
- General logic
  - M00 = Program stop
  - M30 = End of program
  - M09 = Shutter ON
- Plane selection
  - G17 = xy plane
  - G18 = xz plane

  - G19 = yz plane
- Movement
  - G00 X10 Y10 = Rapid positioning $\langle x, y \rangle$
  - G01 X10 Y10 F200 = Linear Interpolation $\langle x, y \rangle$ Feed Rate
  - G01 X10 Y10 I0 J-5 = Circular clockwise Interpolation $\langle x, y \rangle$ $\langle x, y \rangle$ offset relative to end

Bibliography

[1] T. C. Book, “Cargo Targets.”

[2] T. R. P. Language, “Linkage.”