Rapid Motor Characterization Using ODrive Motor Controller

Master's Plan B Project Report

Department of Mechanical Engineering, University of Minnesota

April 9, 2021

Emily R. Goldberg[a]

Adviser/Committee Chair: Professor Tim Kowalewski[a]

Committee Members: Professor Will Durfee[a], Professor Junaed Sattar[b]

[a]Department of Mechanical Engineering, University of Minnesota-Twin Cities, Minneapolis, MN 55455

[b]Department of Computer Science, University of Minnesota-Twin Cities, Minneapolis, MN 55455

Abstract

Brushless DC motors (BLDCs) are increasingly surpassing brushed DC motors in robotics due to their superior power-to-weight ratio, smoothness of torque control, heat dissipation capacity, and nuanced control capabilities. The traditional drawbacks of increased control complexity and wiring are being overcome with recent hardware advances that greatly lower barriers to adoption – one such project is the ODrive, a generalized motor controller that makes it possible to easily switch between motors of different brands by avoiding brand-specific interfaces. However, challenges remain, such as the time-consuming and unreliable process of manual tuning. The key alternative to manual tuning is motor system (plant) identification, but the ODrive is missing two features key to this process: it lacks both the ability to measure and record motor signals in real time, and the ability to send arbitrary test signals. This project adds these two capabilities to make the ODrive easier to deploy with any brushless motor. Voltage test inputs were added to the ODrive firmware by creating a new state machine state that makes use of parameters set in a user-editable configuration struct; a ring buffer was added to store voltage inputs and encoder estimates of motor position and velocity; and a new utility method was added to the ODrive's Python interface to run the test input, pull the data from the ring buffer via USB, and export them to CSV. Data was successfully recorded at a sample rate of approximately 200Hz, and plotting in MATLAB allowed for quick, accurate characterization of a motor via classical controls analysis. This provides a straightforward option for future students to more easily experiment with and cleanly control BLDCs, whether that be with simple PID systems or arbitrarily sophisticated controllers.

Introduction

Traditional robotics typically relies on electric motors for actuation, but which motors – and which control method – depends heavily on application. In recent years, brushless DC motors (BLDCs) have come to greater prevalence in robotic applications due to their unique combination of performance and cost-efficacy. Servos are high-quality, but can be prohibitively expensive; steppers can replace them in some applications, but don't perform well at high speeds; and brushed DC motors (which use a commutator and brushes to convert DC current to AC) are simple and cheap, but arcing on the brush-commutator surface generates significant waste heat, wear, torque ripple, and electromagnetic interference [1]. BLDCs, by replacing the commutator-brush system with rotor position feedback (usually via Hall sensors or encoders), avoid arcing and achieve significantly better performance in speed, efficiency, reliability, and lifespan, with the added guarantee of no torque ripple or deadzone [1], [2]. They are more expensive than brushed DCs due to their need for a built-in driver circuit [2], but are able to achieve a combination of speed variability and good position control at a lower cost than most high-quality alternatives [1].

Good performance, however, is dependent on good control. BLDCs require more complex control than their brushed counterparts, and understanding control options requires an understanding of BLDC design. A BLDC is, at its simplest, composed of a single permanent magnet (the rotor) rotating in the center of three fixed windings spaced 120 degrees apart (the stator) [3]. Each winding current generates a magnetic field, represented for convenience as a "current space vector" with direction equal to the field and magnitude proportional to the current (Fig. 1); the three vectors sum to a net magnetic field which generates torque in the rotor by attracting/repelling the rotor magnetic field [3].

For any given rotor position, there is some optimal current space vector that will generate maximum torque: the vector that is directly orthogonal to the rotor field, i.e. orthogonal to the "direct" (d) axis and parallel to the "quadrature" (q) axis (Fig. 1). This maximizes orthogonal forces (which generate torque) and minimizes parallel forces (which generate wear and waste heat) [3]. Since the three windings are spaced 120 degrees apart from each other, in order to perfectly generate this ideal current space vector as the rotor turns, each winding should theoretically have current varying sinusoidally over time and phase-offset 120 degrees from its two neighbors [3].

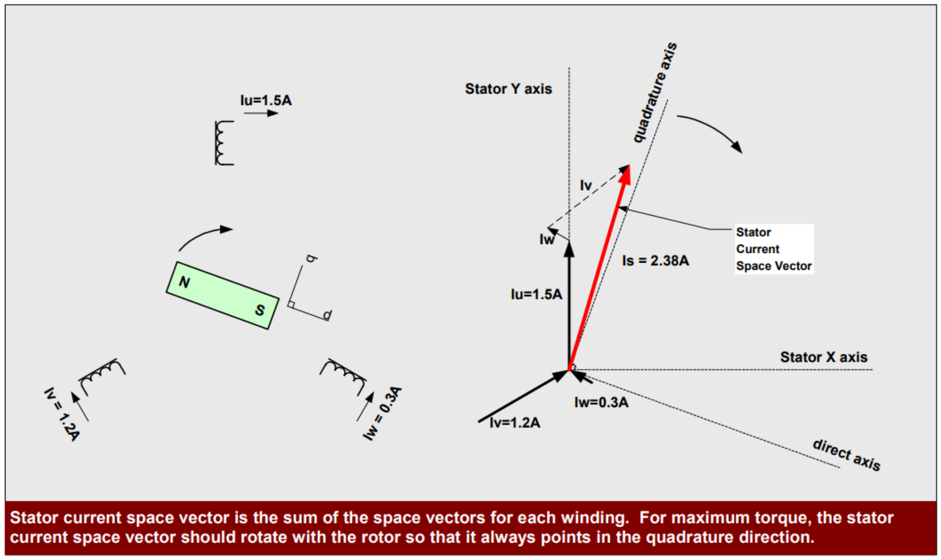

Figure 1. “d-q” coordinate frame of field-oriented control, taken from [3].

The problem, then, is how to calculate the winding currents to generate this optimal sinusoid – a problem addressed using different methods of “electronic commutation” (Fig. 2). Common options include trapezoidal commutation, sinusoidal commutation, and field-oriented control, each of which approximates the ideal sinusoid in a different way. Trapezoidal (or “block”) commutation is the simplest option, approximating the sinusoid by setting each winding either “off” (zero) or “on” ($\pm$ some fixed magnitude) [2]. It is useful in its simplicity, but can be inefficient because it is only capable of producing a current space vector in one of six directions. This means the vector may be misaligned with the ideal vector by up to 30 degrees, causing significant torque ripple and making it difficult to control precisely at slow speeds [3]. In contrast, sinusoidal commutation uses rotor position feedback to generate a true sinusoidal current in each winding [2], achieving close to the ideal current space vector and eliminating most torque ripple. However, its control loop cannot keep up with high motor speeds, limiting its utility in some applications [3]. This leads us to the final commutation form, field-oriented control (FOC). FOC maintains the benefits of sinusoidal commutation, but overcomes its speed limitations by performing its control in a different coordinate system which allows the desired space vector to be constant (the "d-q" reference frame; see [3]). This does require many coordinate transformations (Park and Clarke transforms; see [4]), but these calculations can be done rapidly during operation. By insulating the controller from time-varying effects, FOC therefore achieves performance that is mostly unaffected by motor speed. By blending the low-speed smoothness of sinusoidal commutation and high-speed efficiency of trapezoidal commutation, it performs well as a general-purpose control method for BLDCs in complex applications [3] and allows designers to take advantage of the cost-efficacy of BLDCs while still achieving high-quality control.

However, there are challenges in everyday use of BLDCs, particularly in experimental settings where it may be desirable to quickly switch between different motors. Firstly, different brands often have proprietary user interfaces for the motor's built-in drive circuit; changing between motors of different brands may therefore require substantial reworking of the software for any generalized system. Furthermore, motor tuning (that is, choosing controller gains to ensure good performance) is a nontrivial problem even with relatively simple controllers. Manual tuning (i.e. starting with low gains and iteratively increasing them by trial and error [5]) is feasible, but time-consuming and unreliable [6]. Much more precise control can be achieved by first characterizing the motor (analyzing motor position and velocity response to a

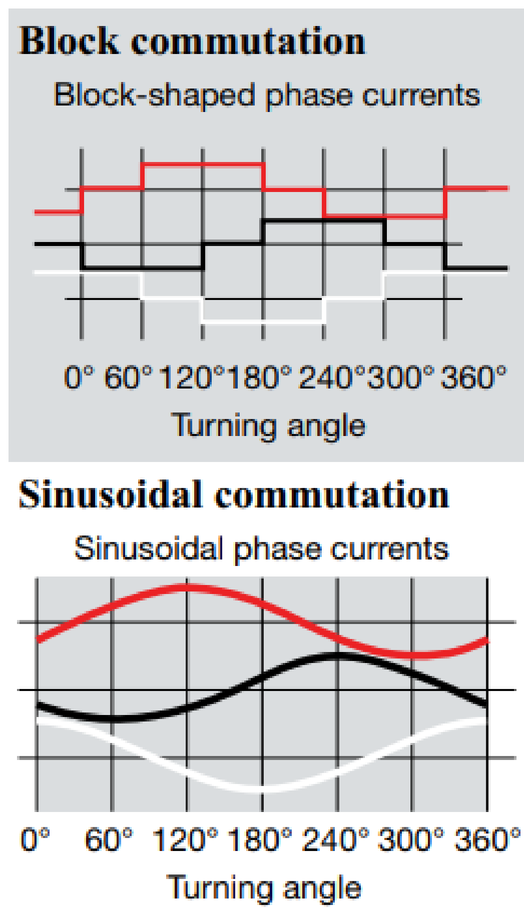

Figure 2. Common methods of commutation, adapted from figure in [2]. Three curves correspond to the three motor winding currents.

known voltage input to produce a “plant” or motor model; sometimes called “system identification” or “motor parameter estimation”), and then designing controller gains to achieve the desired response.

To enable this quick motor characterization, we have identified an open-source generic motor controller, the ODrive, which is electrically compatible with a majority of BLDCs regardless of brand. However, at present the ODrive still requires manual tuning, as it is not equipped with two capabilities necessary for motor characterization. This project therefore sought to edit the ODrive firmware and software to add 1) high-resolution real-time recording of motor data (applied electrical input, motion outputs) with bounded or known sample time jitter, and 2) the ability to send known or desired test signals to characterize the motor (step, impulse, chirp, and noise) in the same time sample as the recorded parameters, while maintaining user-friendliness by exporting that data to human-readable ASCII text files easily parsed by popular programs like MATLAB or Microsoft Excel. Together, these capabilities would make it possible to rapidly characterize any ODrive-compatible motor and ease the experimentation process in the MRD lab or elsewhere.

Materials and Methods

Materials

This project used an ODrive (hardware v3.6-24V) to control a maxon brushless DC motor with encoder (Fig. 3), editing firmware version v0.4.11 of the ODrive project [7] to add in the required features.

ODrive

The ODrive is a motor driver designed to control up to two brushless DC motors (full documentation found at [5]). It is a 5x14cm board (3cm tall if purchased with connectors already soldered on) which can be powered by power supply or battery, available in either 24V and 48V versions. Its hardware is no longer open-source as of hardware v3.6, but earlier schematics are still available for reference [7]. The ODrive is equipped for various forms of interfacing, including USB, UART, and CAN; USB is the slowest, but comes with a user-friendly Python application `odrivetool` that includes useful utility functions for parameter access, error checking, and data plotting. If USB speeds are acceptable, this option is by far the easiest to work with.

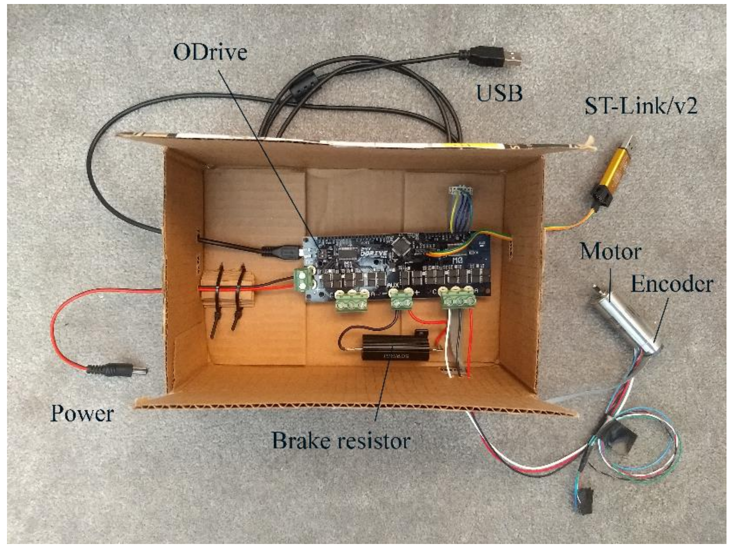

Figure 3. Experimental setup. ST-Link/v2 is used only for flashing new firmware, USB for data collection. This version of ODrive requires 24V power. A simple cardboard jig was fashioned to faciliate board development and debugging while off campus due to the COVID-19 pandemic.

ODrive firmware is entirely open-source [7], and the documentation includes a guide to editing and troubleshooting using VS Code (note that this requires an ST-Link/v2) [8]. Technical support and the developer community are hosted on a forum site and Discord server. The firmware is complex, but this project was possible focusing mainly on the `Axis` class, with minor adjustments to `main.cpp`, `odrive_main.h`, and `communication.cpp` as needed.

The basic ODrive control framework involves two `Axis` objects that are created on startup, each of which controls one motor and encoder by running a state machine. The user may set the `requested_state` axis field, which causes the state machine to call the appropriate `Axis` function and make use of the `Encoder`, `Motor`, and `Controller` class objects as needed.

Motor and encoder: This project was developed and tested using a direct drive (no gearbox) maxon 323218 motor (125 g weight; 45.1 mNm nominal torque; 16 N maximum radial load; 90 W assigned power rating), which comes with a built-in maxon 575828 encoder (32768 counts per revolution). In general, the ODrive will work with most brushless DC motors, and most encoders with bandwidth greater than 200Hz [9]. At time of writing, the ODrive documentation provides spreadsheets with example analysis of useable motors [10] and encoders [9].

When using a particular motor and encoder, the ODrive must be configured with the motor nominal voltage (V); pole pairs (integer); speed constant K_v (rpm/V); and maximum speed (rpm), and the encoder maximum speed (rpm) and CPR (counts per rotation, integer). Note in particular that terminology for CPR is inconsistent and highly brand-dependent. CUI Devices uses pulses per revolution (PPR), which is one fourth of CPR; other manufacturers may use lines per revolution (LPR), which is the same; and some use the same CPR acronym to mean "cycles per revolution", which is actually equivalent to PPR and LPR [11]. Fortunately, it is not harmful to configure the wrong encoder CPR when testing the ODrive. An incorrect CPR setting will cause the motor to spin out of control upon calibration, but the ODrive will shut it down safely and throw an `ERROR_CPR_OUT_OF_RANGE`. If this happens, the most likely adjustments are by factors of x4 or x2. For a guide to setting up the ODrive with a new motor, see the official documentation [5] or the walkthrough specific to this project [12].

Firmware – Embedded C/C++ and CMSIS-RTOS Running ODrive Microcontroller

For this project, a new method `run_motor_characterize_input()` was added to the `Axis` class. To manage it, a new state `AXIS_STATE_MOTOR_CHARACTERIZE_INPUT` was added to the `Axis` class state machine; when `requested_state_` is set to `AXIS_STATE_MOTOR_CHARACTERIZE_INPUT`, the state machine performs checks and then calls `run_motor_characterize_input()`. All parameters needed for `run_motor_characterize_input()` have been added to a new user-editable `Axis` class struct `input_config_` (of custom type `InputConfig_t`); there is one `input_config_` object for each of the two `Axis` objects, and both have been incorporated into `main.cpp` such that they are saved alongside the pre-existing system configurations.

The ODrive is not generally equipped for user-directed voltage control; all standard interface methods assume that the user is using either position, velocity, or current control. However, there is a motor setting for gimbal motors that, when activated, has the sole effect of treating all current control instead as voltage control. This is discouraged due to the risk of damaging the motor, but, if used carefully, it can be used to send raw voltage commands. `AXIS_STATE_MOTOR_CHARACTERIZE_INPUT` was therefore designed to require that the motor be set to `MOTOR_TYPE_GIMBAL` and the control mode to `CTRL_MODE_CURRENT_CONTROL` before running `run_motor_characterize_input()`.

`run_motor_characterize_input()` was constructed for this project based on the ODrive methods that manage calibration and closed-loop control. It runs two control loops in sequence: one to wait for the designated delay time, then another to send the test input. On each timestep, it records time, voltage, position, and velocity data to a ring buffer `motorCharacterizeData` using a new method `record_motor_characterize_data()`. Individual elements are described in detail below:

Control loop management: Like other axis state functions, the new `run_motor_characterize_input()` is designed to use the existing `Axis` method `run_control_loop()` to safely manage the motor. `run_control_loop()` takes an update handler (which must return a Boolean) as its argument and cyclically calls that handler at 8kHz (guaranteed by a hardware timer interrupt in deterministic real-time operating system CMSIS-RTOS). Before each handler call, it updates encoder estimates and performs various system checks to ensure safe motor operation. If any checks fail, it breaks the loop and sets the motor to idle; otherwise, it continues until the update handler returns false.

Time: Project data is timestamped using the existing `Axis` field `loop_counter_`, which is initialized at ODrive startup and then increments upon each cycle of `run_control_loop()` (8kHz) for the lifetime of the state machine. `run_motor_characterize_input()` records the value of `loop_counter_` at start, then uses that initial value to zero subsequent time measurements for data recording.

Voltage limit: `run_motor_characterize_input()` defines a voltage limit using the pre-existing `Motor` class method `effective_current_lim()`. When motor type is set to `MOTOR_TYPE_GIMBAL` (required for voltage control), `effective_current_lim()` returns a voltage limit rather than a current limit, selecting the lowest of the user-configured current limit; $0.56 * [bus\ voltage]$; and a regularly-updated thermal current limit.

Voltage commands: `run_motor_characterize_input()` calculates voltage commands as a function of time for the selected input type and parameters (Table 1). The command is capped if it exceeds the voltage limit.

Input Type	**Voltage command as a function of time (all parameters from `input_config_`)**
Step	$V(t) = step\_voltage$
Impulse	$V(t) = \begin{cases} impulse\_voltage\ if\ [steps\ elapsed] < impulse\_peakDuration \\ 0\ otherwise \end{cases}$
Noise	$V(t) \in [-n, n]$ Where $n = \left(\frac{noise\_max}{100} * voltage\_lim\right)$ for $noise\_max\ \mathbb{Z} \in [1\ 100]$
Exponential chirp	$V(t) = chirp\_amplitude * \sin(phase) + chirp\_midline$ Where $phase = 2\pi * chirp\_freqLow * \left(\frac{k^{x*test\_duration} - 1}{\log(k)}\right)$ for $k = \left(\frac{chirp\_freqHigh}{chirp\_freqLow}\right)^{\frac{1}{test\_duration}}$

Table 1. Calculation of voltage commands using saved configuration parameters.

Encoder readings: Up-to-date encoder estimates of motor phase (in electrical radians), position (in encoder counts) and velocity (in counts/s) are obtained by accessing the latest `Encoder` class fields `pos_estimate_` and `vel_estimate_` (part of standard ODrive architecture). These estimates are updated each cycle (8kHz) by `run_control_loop()`. Note that the `Encoder` class field `phase` refers to the latest estimate of motor position in electrical radians, relative to the index position; contrast with `pos_estimate_`, which is in encoder counts and does not wrap unless specifically configured to do so.

Motor updates: `run_motor_characterize_input()` sends voltage commands to the motor using the standard `Motor` class method `update()`. `Motor::update()` requires either 1) a constant voltage and target phase/phase velocity, or 2) a target voltage and latest observed phase/phase velocity. Since this application requires a target voltage, the other arguments must be provided by fetching the latest encoder velocity estimate and using it to estimate phase velocity:

$$phase\_vel = 2\pi\left(\frac{vel\_estimate}{cpr * pole\_pairs}\right)\left[\frac{electrical\ rad}{s}\right]$$

`Motor::update()` passes along the command to low-level methods that generate the appropriate field-oriented control voltages and add them to the queue of motor phase timings.

Data collection: Characterization data is stored in a custom 4x128 ring buffer `motorCharacterizeData` (which is modeled on the existing ODrive `oscilloscope` buffer). It is defined globally so that it can be accessed by both `axis.cpp` (for data writing) and `communication.cpp` (for user-accessibility). On each loop, `run_motor_characterize_input()` calls custom method `record_motor_characterize_data()` with the latest timestep and voltage command, and `record_motor_characterize_data()` writes them to `motorCharacterizeData` along with the latest encoder estimates of motor position and velocity.

Data accessibility: `motorCharacterizeData` can be accessed by index in `odrivetool` using a set of four get functions (`get_motorCharacterizeData_timestep()`, `get_motorCharacterizeData_voltage()`, etc.) because these functions were added to the communication protocol in `communication.cpp`. At any given time, the "latest recorded observation" can be accessed by calling the get functions on custom index `motorCharacterizeData_pos` (which is also user-accessible).

Software – User Host Computer Python Code

The ODrive itself has insufficient memory for large data files, so the USB user interface `odrivetool` was edited to add a new method `run_motor_characterize_input()` that runs the input, pulls characterization data from the firmware buffer, and exports it to CSV. Taking arguments odrv (ODrive object, default `odrv0`), axis (0 or 1), and export directory (string), `run_motor_characterize_input()` sets the `requested_state` of the specified axis to `AXIS_STATE_MOTOR_CHARACTERIZE_INPUT` (triggering the state machine to begin the test input), and then continuously submits pull requests for the "latest recorded" sample values of `motorCharacterizeData` until the input ends.

Note that `odrivetool` can only be used when connecting over USB. This has drawbacks, since the connection is slow, but `run_motor_characterize_input()` has been designed to mitigate this as much as possible by saving all data to an array and only writing to CSV once data collection is finished.

<u>Motor Data and Basic Plant Model</u>

Data is formatted such that the CSV can be loaded directly into MATLAB, where it is plotted and analyzed. Simple plot analysis can be used to identify the time constant $\frac{1}{a}$ (e.g. by finding the settling time T_s and taking $a = \frac{4}{T_s}$), steady-state value SSV, and gain $K = SSV * a$, and construct a typical DC motor plant function $G(s) = \frac{K}{s(s+a)}$.

Results & Discussion

Firmware and software edits successfully achieved reasonably high-resolution data collection using the specified test inputs. The user may edit `<axis>.input_config` in `odrivetool`, then run the selected test input and export the data to a local directory using `run_motor_characterize_input()` with arguments ODrive object (e.g. default odrv0), axis number (0 or 1), and export directory (string). Data files are formatted such that they can be loaded into MATLAB for plotting and analysis (Fig. 4).

Sample rate was limited by USB connection and ODrive communication protocol standards. Standard ODrive firmware communication protocols do not allow for export of more than a single value on one call, so `run_motor_characterize_input()` was designed to access the "last-recorded index" `motorCharacterizeData_pos` on each loop, then run four individual get functions (`get_motorCharacterizeData_timestep()`, `get_motorCharacterizeData_voltage()`, `get_motorCharacterizeData_position()`, and `get_motorCharacterizeData_velocity()`) for that index. This guarantees that all four pieces of data are from the same timestep, but also means that data recording can only capture “snapshots” of the recorded data.

By optimizing `run_motor_characterize_input()`, it was possible on this test system to achieve an average sample rate of 200Hz (Fig. 5). Since firmware-side data is recorded at 8kHz, this means data exporting is capturing approximately one out of every forty observations. Further improvements could possibly be made with more targeted work specifically on ODrive communication methods. This project accessed the ring buffer using the standard set of `make_protocol_XXX()` functions (described in the ODrive’s `protocol.hpp`; must be called at startup in order to allow the user to access the specified firmware field); these protocol methods can act only on a limited set of data types (e.g. float, bool), and do not allow for custom structs that could contain multiple pieces of data. If the protocols could be modified to allow the user to pull all data with a single struct call, this might substantially increase the sampling rate. On the other hand, there is also the simple hardware limitation of USB speed. The ODrive does allow for other communication protocols (UART, CAN, etc.), which would undoubtably be faster; however, this would require an entirely new user interface, as `odrivetool` is limited to USB.

Despite these limitations, sampled data was sufficiently high-resolution to allow motor characterization, since the bandwidth of the BLDC motor is well below kHz sampling rates. Based on velocity step response for a quarter-volt step, characterization analysis produced a plant transfer function $G(s) = \frac{1502}{s(s+333)}$ with time constant $\frac{1}{a} = \frac{1}{333.33} = 0.003\ s$; steady-state velocity $SSV = 4.5060 \frac{rev}{s}$; and gain $K = SSV * a = 4.5060 \frac{rev}{s} * \ 333.33\ s^{-1} = 1502.0 \frac{rev}{s^2}$ (Fig. 6). For example data and MATLAB analysis code, see the *docs/references* folder on the lab GitHub [13].

291

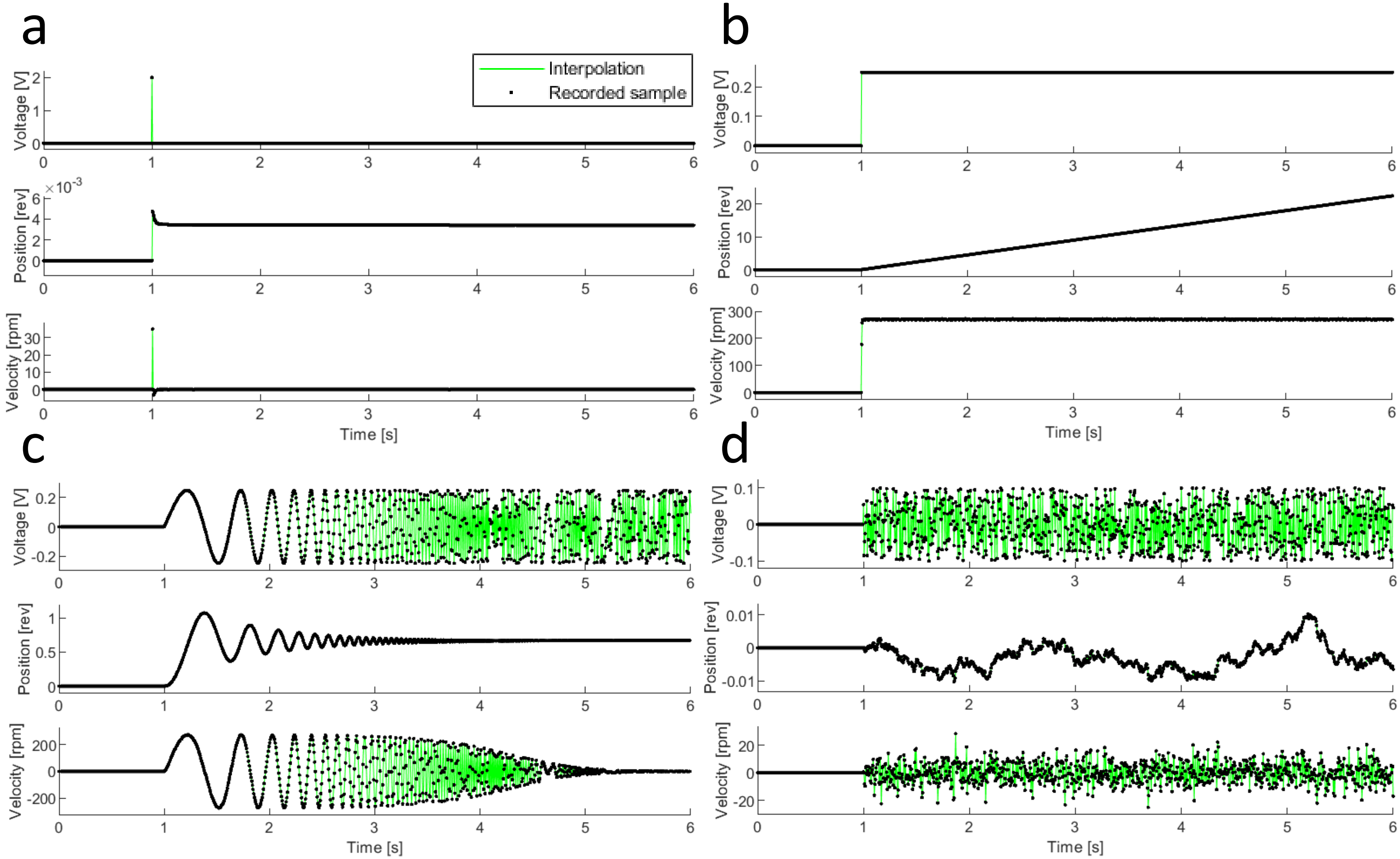

Figure 4. Actual motor data collected from the four newly-added test signals. Impulse (a), step (b), exponential chirp (c), and white noise (d) voltage inputs with encoder-recorded position and velocity responses.

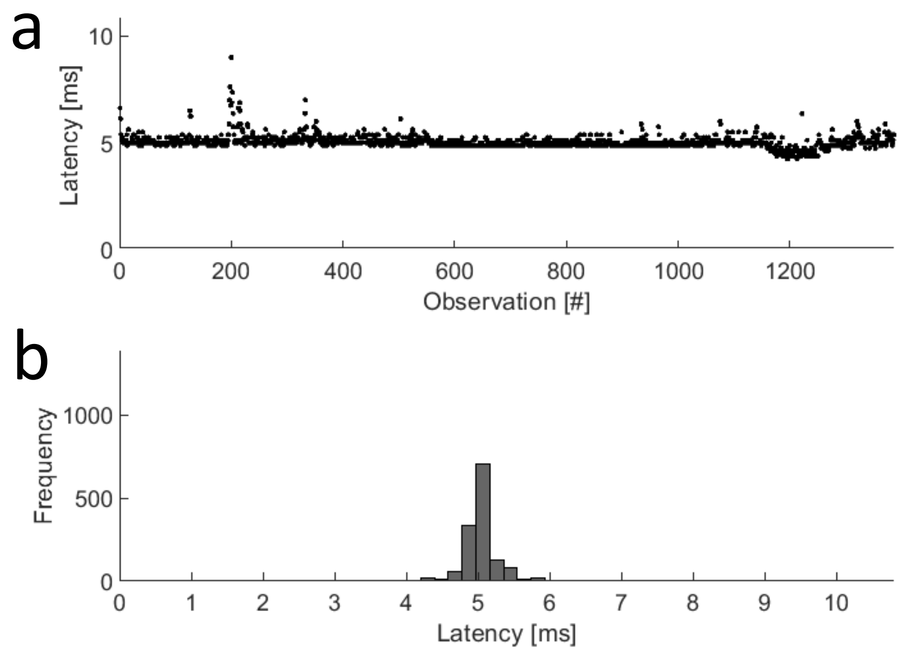

Figure 5. Typical Python-side latency for data collection (mean 5.0 ms) by sample (a) and histogram (b).

Using this transfer function, it should be possible to control the motor simply by deriving the transfer function of the controller $C(s)$, finding $T(s) = G(s)C(s)$, and performing state space pole placement with $T(s)$. Note, however, that finding $C(s)$ may be nontrivial depending on the controller in question. The ODrive uses a combination of linked P and PI loops for position, velocity, and current control (see Appendix I for example analysis), of which the position and velocity loops are user-facing and the current loop is not [5]. Alternatively, the user might choose to introduce their own controller, though this would require editing the firmware.

Finally, it is important to note that, while this project was nearing completion, the ODrive firmware and (to a lesser extent) `odrivetool` were updated from version 4 to version 5 and have changed in some ways that could benefit the project. Firstly, v0.5 completely overhauls the communication protocol system; depending on what exactly has been changed (e.g. possible to pull custom structs, multiple values at once), it may be possible to speed up the sampling rate and improve

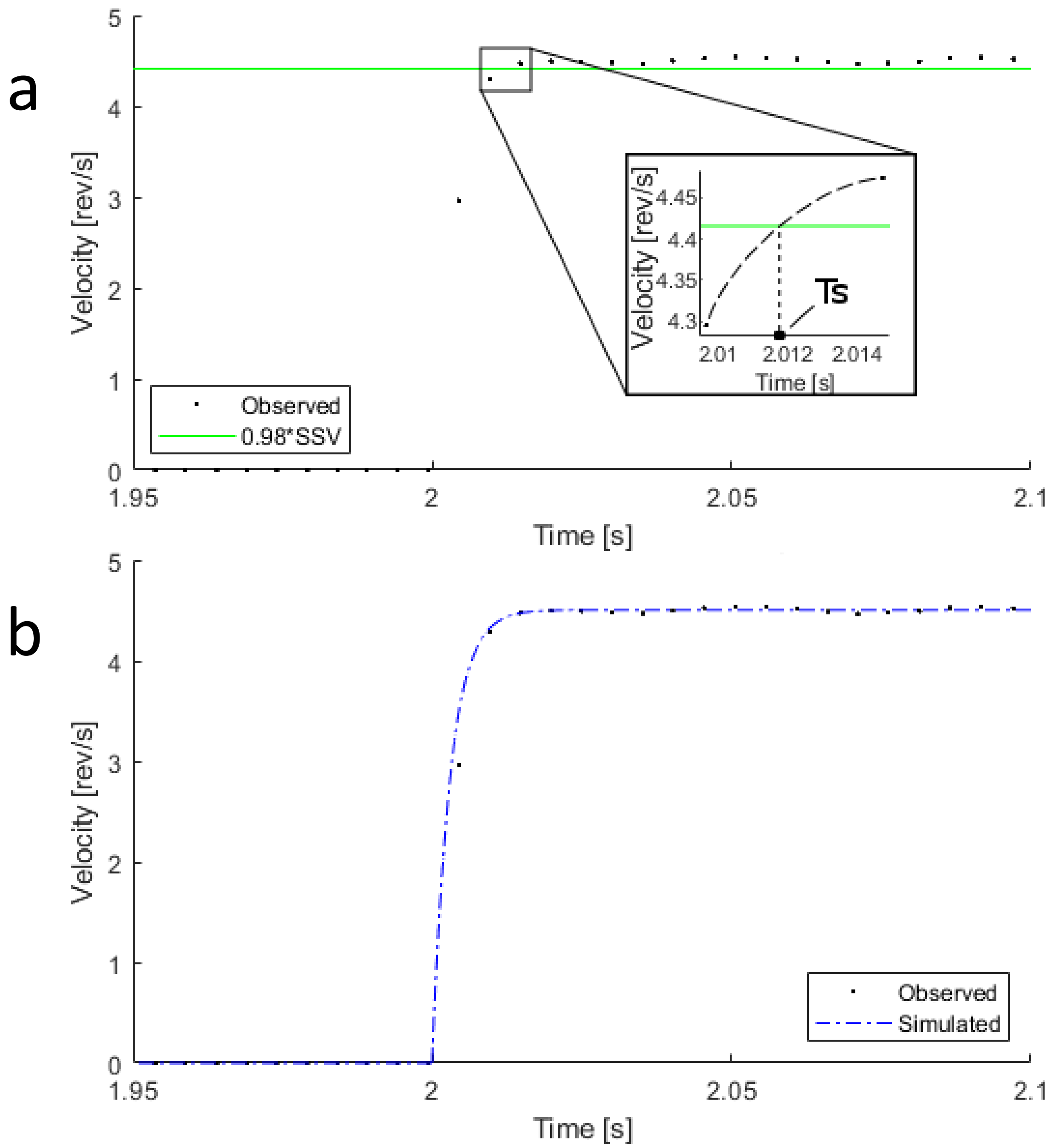

Figure 6. (a) Method to derive T_s from a step response. Plot the within-2% steady-state value $0.98 * SSV$; estimate where the step response crosses that line; and take that crossing time as T_s. From here, the transfer function can be estimated, in this case producing a simulated step response (b) using $G(s) = \frac{1502}{s(s+333)}$.

the resolution of the recorded response signal. Secondly, v0.5 has a new `odrivetool` class `BulkCapture`, which has some similarity to the data capture aspects of this project but uses threading to run in the background, which could be useful if the user wishes to collect data during more complicated behavior (e.g. running multiple input types in sequence). Finally, of course, forward-compatibility would be beneficial in its own right, since ODrive firmware is under active development and new capabilities are added frequently. That said, updating this project to v0.5 would be a nontrivial task, as major changes were made to the communication protocol system; whether the time investment is worthwhile will therefore depend on application, success of the project as-is, and the value placed on remaining current with ODrive updates.

Conclusions

This project successfully enabled motor characterization by adding test input and data export features to ODrive v0.4 firmware and software, though its sampling resolution was limited to 200Hz by software and hardware restrictions. Future work would be useful to update these methods to v0.5 to take advantage of advances in communication methods. Such a revision would require confirming the continued viability of firmware-side test input methods in v0.5; researching the capabilities of v0.5 communication and optimizing as appropriate; and potentially making use of the new `BulkCapture` class for `odrivetool` data recording. In the interim, note that it should be possible to characterize the motor using this project's v0.4 firmware, then switch to v0.5 firmware if desired. This document is intended to serve as a reference for practical use at the University of Minnesota or elsewhere, and to that end all project code may be found on a public GitHub repository[1] [13], including MRD-lab-specific guides to ODrive setup [12] and development [14].

Acknowledgements

The author would like to thank Professor Tim Kowalewski for his advice and support, as well as Yusra Farhat Ullah and Mark Gotthelf for their work on concept development and project goals. Thanks are also due to ODrive developers Oskar Weigl and Paul Guenette, who were very open to providing feedback and assistance for design questions and clarifications.

---

[1] Search "ERG" within *Firmware* and *tools* to find all edits specific to this project.

This work was supported, in part, by the National Science Foundation CAREER Grant under Award No. 1847610. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the National Science Foundation.

References

[1] J. Zhao and Y. Yangwei, “Brushless DC Motor Fundamentals Application Note,” *MPS*. Monolithic Power, pp. 1–19, 2014.

[2] J. Braun, “Formulae Handbook,” *maxon academy*. maxon motor, Sechseln, pp. 1–60, 2012.

[3] C. Rollman, “What is ‘Field Oriented Control’ and what good is it?,” *Copley Controls Corp.* pp. 1–13, 2002.

[4] “Park, Inverse Park And Clarke, Inverse Clarke Transformations MSS Software Implementation User Guide,” *Microsemi Corporation*. Alieso Viejo, CA, pp. 5–9, 2013.

[5] “ODrive Documentation,” *ODrive Robotics*, 2021. [Online]. Available: docs.odriverobotics.com. [Accessed: 01-Apr-2021].

[6] “Motion System Tuning.” Rockwell Automation, Milwaukee, WI, pp. 22–23, 2020.

[7] “ODrive Robotics GitHub,” *ODrive Robotics*. [Online]. Available: github.com/odriverobotics. [Accessed: 01-Apr-2021].

[8] “ODrive Firmware Developer Guide,” *ODrive Robotics*. [Online]. Available: docs.odriverobotics.com/developer-guide. [Accessed: 01-Apr-2021].

[9] “ODrive encoder guide,” *ODrive Robotics*. [Online]. Available: https://docs.google.com/spreadsheets/d/1OBDwYrBb5zUPZLrhL98ezZbg94tUsZcdTuwiVNgVqpU/edit#gid=0. [Accessed: 01-Apr-2021].

[10] "ODrive motor guide," *ODrive Robotics*. [Online]. Available: https://docs.google.com/spreadsheets/d/12vzz7XVEK6YNIOqH0jAz51F5VUpc-lJEs3mmkWP1H4Y/edit#gid=0. [Accessed: 01-Apr-2021].

[11] R. Smoot, "What's the Difference Between an Incremental Encoder's PPR, CPR, and LPR?," *CUI Insights: Motion*, 2021. [Online]. Available: cuidevices.com/blog/what-is-encoder-ppr-cpr-and-lpr. [Accessed: 01-Apr-2021].

[12] E. Goldberg, "ODrive Setup Walkthrough." MRD Lab, University of Minnesota Department of Mechanical Engineering, Minneapolis, MN, 2021.

[13] "MRD Lab GitHub," *Medical Robotics and Devices Lab*. [Online]. Available: github.com/labmrd. [Accessed: 01-Apr-2021].

[14] E. Goldberg, "VS Code for ODrive." MRD Lab, University of Minnesota Department of Mechanical Engineering, Minneapolis, MN, 2021.

Appendix I. ODrive Controller Simplification

In order to optimize controller gains for a given motor transfer function, it is necessary to know the transfer function for the controller. Standard ODrive control uses a series of P and PI loops for position control [5], of which only the position and velocity controller gains are user-settable.

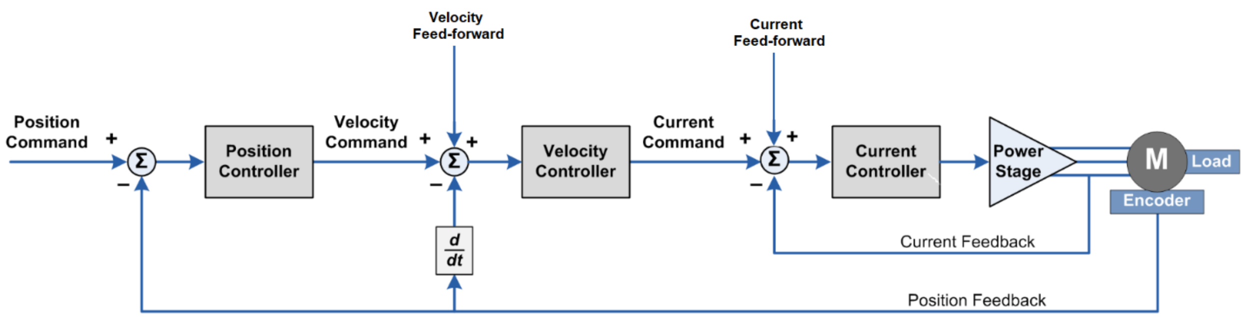

Figure A1. ODrive position controller; image taken from ODrive documentation [5].

Making the assumption that feed-forward terms are zero and that the current control loop can be represented as a simple PI transfer function $I(s)$, we may use block diagram simplification methods (Fig. A2) to derive an overall controller transfer function:

$$C(s) = \frac{\frac{P(s)I(s)V(s)}{1 + I(s)V(s)s}}{1 + \frac{P(s)I(s)V(s)}{1 + I(s)V(s)s}}$$

$$= \frac{P(s)I(s)V(s)}{1 + I(s)V(s)s + P(s)I(s)V(s)}$$

Where $P(s) = K_p$, $V(s) = K_v + \frac{K_{vi}}{s}$, and $I(s) = K_i + \frac{K_{ii}}{s}$.

K_p, K_v, and K_{vi} may be defined by the user by setting `<odrive>.controller.config` members `pos_gain`, `vel_gain`, and `vel_integrator_gain`, respectively. The current control loop, on the other hand, is part of the Motor class, where K_i and K_{ii} correspond to the `p_gain` and `i_gain` members of `<odrive>.motor.current_control`. `p_gain` and `i_gain` are not user-configured; they are instead set

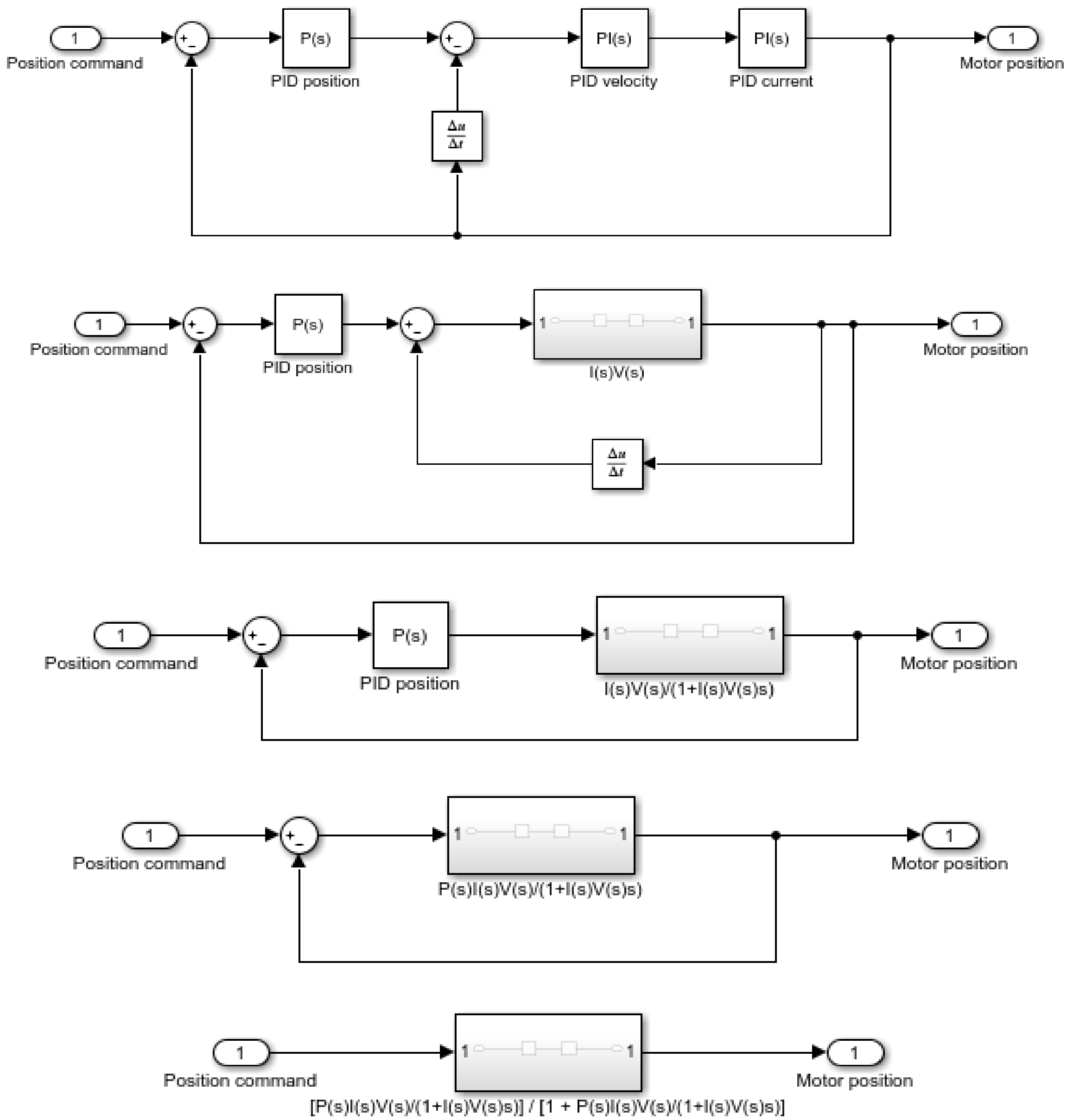

Figure A2. Block diagram simplification of ODrive controller.

automatically upon motor calibration as a function of the configured current control bandwidth and the measured motor phase resistance and phase inductance:

$$K_i = current\_control\_bandwidth * phase\_inductance$$

$$K_{ii} = \left(\frac{phase\_resistance}{phase\_inductance}\right) * K_i$$

If phase inductance and phase resistance are reasonably constant, it should be possible to treat K_i and K_{ii} as observable constants for the purposes of control design.

Appendix II. Summary of Contributions

A list of all modifications made to ODrive v0.4.11. All edits are labeled with an “ERG” comment, so edits may easily be viewed in the code [13] by searching “ERG”.

<u>Firmware:</u> Input commands and data collection

I. Test input
- a. axis.cpp and axis.hpp
  - i. Added input configuration to Axis class
    1. Added enum definition InputType_t
    2. Added struct definition InputConfig_t
    3. Added input_config argument (of type InputConfig_t) to Axis class
    4. Add input_config_ field (of type InputConfig_t)
    5. Edited make_protocol_definitions() to make input_config_ and its members user-accessible
  - ii. Added new state machine behavior to run test input
    1. Added AXIS_STATE_MOTOR_CHARACTERIZE_INPUT to State_t
    2. Edited run_state_machine_loop() to include behavior for AXIS_STATE_MOTOR_CHARACTERIZE_INPUT
    3. Added method run_motor_characterize_input() (120 lines)
- b. main.cpp
  - i. Added array input_configs
  - ii. Added input_configs to save_configuration() and load_configuration()
  - iii. Added input_config argument to axes initialization

II. Data recording and user-accessibility
- a. odrive_main.h
  - i. Added ring buffer motorCharacterizeData, of size MOTORCHARACTERIZEDATA_SIZE, with index tracker motorCharacterizeData_pos
- b. communication.cpp
  - i. Added initialization for ring buffer
  - ii. Added four static ‘get by index’ functions for timestep, voltage, position, and velocity in motorCharacterizeData
  - iii. Edited make_obj_tree() to make motorCharacterizeData_pos, motorCharacterizeData_size, and the four get functions user-accessible
- c. axis.cpp and axis.hpp
  - i. Added method record_motor_characterize_data() (8 lines), which writes data to motorCharacterizeData at index motorCharacterData_pos when called as part of run_motor_characterize_input()

<u>Python:</u> User interface, data retrieval and export

I. enums.py - added AXIS_STATE_MOTOR_CHARACTERIZE_INPUT

II. utils.py - added method run_motor_characterize_input() (85 lines)

III. shell.py - added run_motor_characterize_input() to launch_shell()