Rapid Motor Characterization Using ODrive Motor Controller

Master's Plan B Project Report

Department of Mechanical Engineering, University of Minnesota

May 3, 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.

2

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

3

72 generate torque) and

73 minimizes parallel

74 forces (which generate

75 wear and waste heat)

76 [3]. Since the three

77 windings are spaced

78 120 degrees apart from

79 each other, in order to

80 perfectly generate this

81 ideal current space

Stator current space vector is the sum of the space vectors for each winding. For maximum torque, the stator current space vector should rotate with the rotor so that it always points in the quadrature direction.

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

82 vector as the rotor turns, each winding should theoretically have current varying sinusoidally over time

83 and phase-offset 120 degrees from its two neighbors [3].

84   The problem, then, is how to calculate the winding currents to generate this optimal sinusoid – a

85 problem addressed using different methods of "electronic commutation" (Fig. 2). Common options

86 include trapezoidal commutation, sinusoidal commutation, and field-oriented control, each of which

87 approximates the ideal sinusoid in a different way. Trapezoidal (or "block") commutation is the simplest

88 option, approximating the sinusoid by setting each winding either "off" (zero) or "on" ($\pm$ some fixed

89 magnitude) [2]. It is useful in its simplicity, but can be inefficient because it is only capable of producing

90 a current space vector in one of six directions. This means the vector may be misaligned with the ideal

91 vector by up to 30 degrees, causing significant torque ripple and making it difficult to control precisely at

92 slow speeds [3]. In contrast, sinusoidal commutation uses rotor position feedback to generate a true

93 sinusoidal current in each winding [2], achieving close to the ideal current space vector and eliminating

94 most torque ripple. However, its control loop cannot keep up with high motor speeds, limiting its utility

95 in some applications [3]. This leads us to the final commutation form, field-oriented control (FOC). FOC
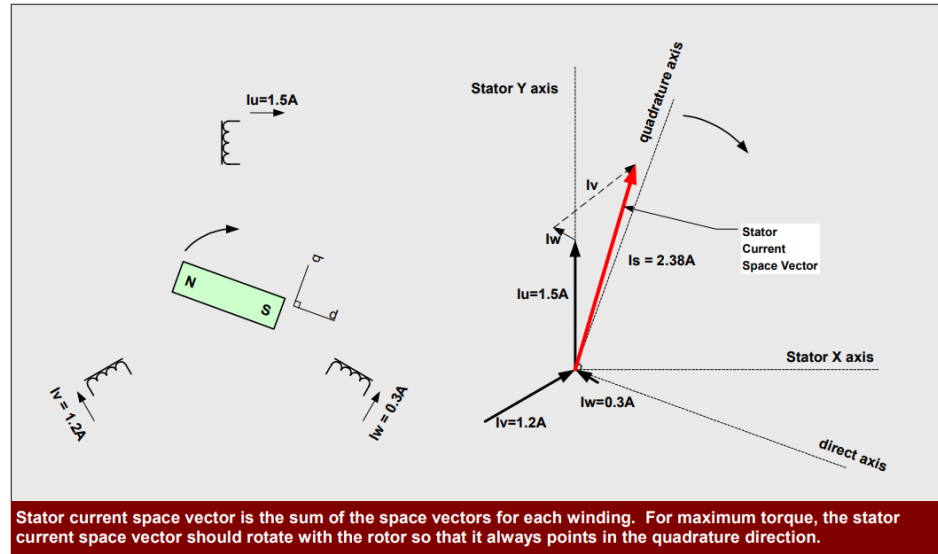
4

96 maintains the benefits of sinusoidal commutation, but overcomes its speed limitations by performing its

97 control in a different coordinate system which allows the desired space vector to be constant (the "d-q"

98 reference frame; see [3]). This does require many coordinate transformations (Park and Clarke

99 transforms; see [4]), but these calculations can be done rapidly during operation. By insulating the

100 controller from time-varying effects, FOC therefore achieves performance that is mostly unaffected by

101 motor speed. By blending the low-speed smoothness of sinusoidal commutation and high-speed

102 efficiency of trapezoidal commutation, it performs well as a general-purpose control method for BLDCs

103 in complex applications [3] and allows designers to take advantage of the cost-efficacy of BLDCs while

104 still achieving high-quality control.

105   However, there are challenges in everyday use of

106 BLDCs, particularly in experimental settings where it may

107 be desirable to quickly switch between different motors.

108 Firstly, different brands often have proprietary user

109 interfaces for the motor's built-in drive circuit; changing

110 between motors of different brands may therefore require

111 substantial reworking of the software for any generalized

112 system. Furthermore, motor tuning (that is, choosing

113 controller gains to ensure good performance) is a

114 nontrivial problem even with relatively simple controllers.

115 Manual tuning (i.e. starting with low gains and iteratively

116 increasing them by trial and error [5]) is feasible, but time-

117 consuming and unreliable [6]. Much more precise control

118 can be achieved by first characterizing the motor

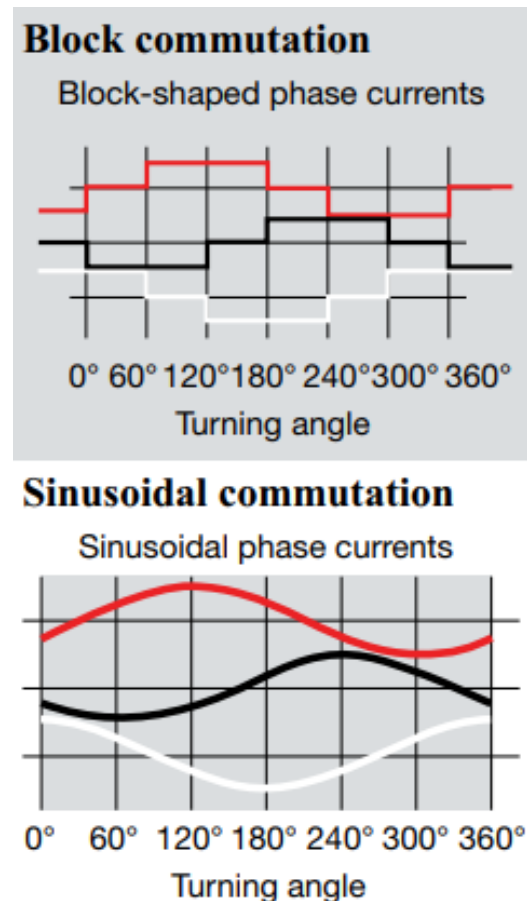119 (analyzing motor position and velocity response to a

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

5

120   known voltage input to produce a "plant" or motor model; sometimes called "system identification" or

121   "motor parameter estimation"), and then designing controller gains to achieve the desired response.

122   To enable this quick motor characterization, we have identified an open-source generic motor

123   controller, the ODrive, which is electrically compatible with a majority of BLDCs regardless of brand.

124   However, at present the ODrive still requires manual tuning, as it is not equipped with two capabilities

125   necessary for motor characterization. This project therefore sought to edit the ODrive firmware and

126   software to add 1) high-resolution real-time recording of motor data (applied electrical input, motion

127   outputs) with bounded or known sample time jitter, and 2) the ability to send known or desired test

128   signals to characterize the motor (step, impulse, chirp, and noise) in the same time sample as the

129   recorded parameters, while maintaining user-friendliness by exporting that data to human-readable

130   ASCII text files easily parsed by popular programs like MATLAB or Microsoft Excel. Together, these

131   capabilities would make it possible to rapidly characterize any ODrive-compatible motor and ease the

132   experimentation process in the MRD lab or elsewhere.

133   Materials and Methods

134   <u>Materials</u>

135   This project used an ODrive (hardware v3.6-24V) to control a maxon brushless DC motor with encoder

136   (Fig. 3), editing firmware version v0.5.1 of the ODrive project [7] to add in the required features.

137   **ODrive**

138   The ODrive is a motor driver designed to control up to two brushless DC motors (full documentation

139   found at [5]). It is a 5x14cm board (3cm tall if purchased with connectors already soldered on) which can

140   be powered by power supply or battery, available in either 24V and 48V versions. Its hardware is no

141   longer open-source as of hardware v3.6, but earlier schematics are still available for reference [7]. The

142   ODrive is equipped for various forms of interfacing, including USB, UART, and CAN; USB is the slowest,

6

143   but comes with a user-friendly Python

144   application `odrivetool` that includes

145   useful utility functions for parameter

146   access, error checking, and data plotting.

147   If USB speeds are acceptable, this option

148   is by far the easiest to work with.

149   ODrive firmware is entirely open-

150   source [7], and the documentation

151   includes a guide to editing and

152   troubleshooting using VS Code (note that

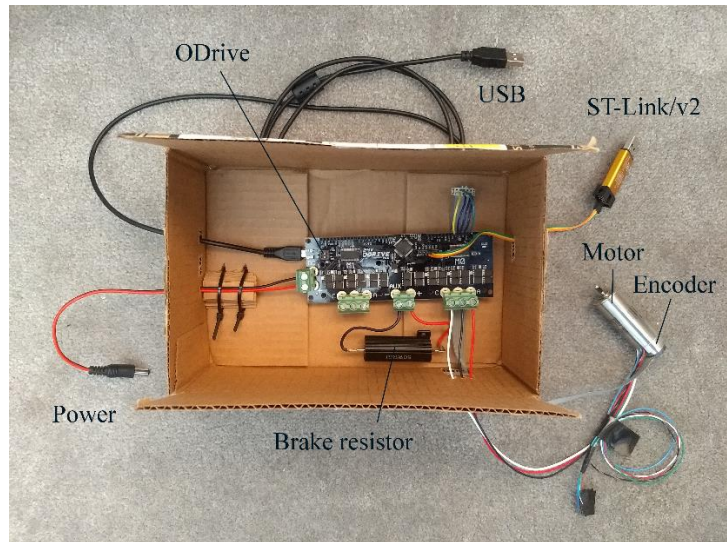153   this requires an ST-Link/v2) [8]. Technical

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.

154   support and the developer community are hosted on a forum site and Discord server. The firmware is

155   complex, but this project was possible focusing mainly on the `Axis` class, with minor adjustments to

156   `main.cpp`, `odrive_main.h`, `communication.cpp`, and `odrive-interface.yaml` as needed.

157   The basic ODrive control framework involves two `Axis` objects that are created on startup, each

158   of which controls one motor and encoder by running a state machine. The user may set the

159   `requested_state` axis field, which causes the state machine to call the appropriate `Axis` function and

160   make use of the `Encoder`, `Motor`, and `Controller` class objects as needed.

161   **Motor and encoder:** This project was developed and tested using a direct drive (no gearbox) maxon

162   323218 motor (125 g weight; 45.1 mNm nominal torque; 16 N maximum radial load; 90 W assigned

163   power rating), which comes with a built-in maxon 575828 encoder (32768 counts per revolution). In

164   general, the ODrive will work with most brushless DC motors, and most encoders with bandwidth

7

165   greater than 200Hz [9]. At time of writing, the ODrive documentation provides spreadsheets with

166   example analysis of useable motors [10] and encoders [9].

167   When using a particular motor and encoder, the ODrive must be configured with the motor

168   nominal voltage (V); pole pairs (integer); speed constant K_v (rpm/V); and maximum speed (rpm), and the

169   encoder maximum speed (rpm) and CPR (counts per rotation, integer). Note in particular that

170   terminology for CPR is inconsistent and highly brand-dependent. CUI Devices uses pulses per revolution

171   (PPR), which is one fourth of CPR; other manufacturers may use lines per revolution (LPR), which is the

172   same; and some use the same CPR acronym to mean "cycles per revolution", which is actually

173   equivalent to PPR and LPR [11]. Fortunately, it is not harmful to configure the wrong encoder CPR when

174   testing the ODrive. An incorrect CPR setting will cause the motor to spin out of control upon calibration,

175   but the ODrive will shut it down safely and throw an `ERROR_CPR_OUT_OF_RANGE`. If this happens, the

176   most likely adjustments are by factors of x4 or x2. For a guide to setting up the ODrive with a new

177   motor, see the official documentation [5] or the walkthrough specific to this project [12].

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

179   For this project, a new method `run_motor_characterize_input()` was added to the `Axis` class. To

180   manage it, a new state `AXIS_STATE_MOTOR_CHARACTERIZE_INPUT` was added to the `Axis` class state

181   machine; when `requested_state_` is set to `AXIS_STATE_MOTOR_CHARACTERIZE_INPUT`, the state

182   machine performs checks and then calls `run_motor_characterize_input()`. All parameters needed for

183   `run_motor_characterize_input()` have been added to a new user-editable `Axis` class struct

184   `input_config_` (of custom type `InputConfig_t`); there is one `input_config_` object for each of the

185   two `Axis` objects, and both have been incorporated into `main.cpp` such that they are saved alongside

186   the pre-existing system configurations.

187   The ODrive is not generally equipped for user-directed voltage control; all standard interface

188   methods assume that the user is using either position, velocity, or current control. However, there is a

189   motor setting for gimbal motors that, when activated, has the sole effect of treating all current control

190   instead as voltage control. This is discouraged due to the risk of damaging the motor, but, if used

191   carefully, it can be used to send raw voltage commands. `AXIS_STATE_MOTOR_CHARACTERIZE_INPUT` was

192   therefore designed to require that the motor type be set to `MOTOR_TYPE_GIMBAL` and the control mode

193   to `CONTROL_MODE_TORQUE_CONTROL` (with `<motor>.config.torque_constant = 1`) before running

194   `run_motor_characterize_input()`.

195   `run_motor_characterize_input()` was constructed for this project based on the ODrive

196   methods that manage calibration and closed-loop control. It runs two control loops in sequence: one to

197   wait for the designated delay time, then another to send the test input. On each timestep, it records

198   time, voltage, position, and velocity data to a ring buffer `motor_characterize_data` using a new

199   method `record_motor_characterize_data()`. Individual elements are described in detail below:

200   **Control loop management**: Like other axis state functions, the new `run_motor_characterize_input()`

201   is designed to use the existing `Axis` method `run_control_loop()` to safely manage the motor.

202   `run_control_loop()` takes an update handler (which must return a Boolean) as its argument and

203   cyclically calls that handler at 8kHz (guaranteed by a hardware timer interrupt in deterministic real-time

204   operating system CMSIS-RTOS). Before each handler call, it updates encoder estimates and performs

205   various system checks to ensure safe motor operation. If any checks fail, it breaks the loop and sets the

206   motor to idle; otherwise, it continues until the update handler returns false.

207   **Time**: Project data is timestamped using the existing `Axis` field `loop_counter_`, which is initialized at

208   ODrive startup and then increments upon each cycle of `run_control_loop()` (8kHz) for the lifetime of

209   the state machine. `run_motor_characterize_input()` records the value of `loop_counter_` at start,

210   then uses that initial value to zero subsequent time measurements for data recording.

211   **Voltage limit**: `run_motor_characterize_input()` defines a voltage limit using the pre-existing `Motor`

212   class method `effective_current_lim()`. When motor type is set to `MOTOR_TYPE_GIMBAL` (required for

213   voltage control), `effective_current_lim()` returns a voltage limit rather than a current limit, selecting

214   the lowest of the user-configured current limit; $0.56 * [bus\ voltage]$; and a regularly-updated thermal

215   current limit.

216   **Voltage commands**: `run_motor_characterize_input()` calculates voltage commands as a function of

217   time for the selected input type and parameters (Table 1). The command is capped if it exceeds the

218   voltage limit.

Input Type	Voltage command as a function of time (all parameters from `input_config_`)
Step	$V(t) = step\_voltage$
Impulse	$V(t) = \begin{cases} impulse\_voltage \text{ if } [steps\ elapsed] < impulse\_peakDuration \\ 0 \text{ } otherwise \end{cases}$
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.

219   **Encoder readings**: Up-to-date encoder estimates of motor phase (in electrical radians), position (in

220   turns) and velocity (in turns/s) are obtained by accessing the latest `Encoder` class fields `pos_estimate_`

221   and `vel_estimate_` (part of standard ODrive architecture). These estimates are updated each cycle

222   (8kHz) by `run_control_loop()`. Note that the `Encoder` class field `phase` refers to the latest estimate of

10

223   motor position in electrical radians, relative to the index position; contrast with `pos_estimate_`, which

224   is in motor turns and does not wrap unless specifically configured to do so.

225   **Motor updates**: `run_motor_characterize_input()` sends voltage commands to the motor using the

226   standard `Motor` class method `update()`. In voltage control mode, `Motor::update()` requires either 1) a

227   constant voltage and target phase/phase velocity, or 2) a target voltage and latest observed

228   phase/phase velocity. Since this application requires a target voltage, the other arguments must be

229   provided by fetching the latest encoder velocity estimate and using it to estimate phase velocity:

230
$$phase\_vel = 2\pi \left(\frac{vel\_estimate}{cpr * pole\_pairs} \right) [\frac{electrical\,rad}{s}]$$

231   `Motor::update()` passes along the command to low-level methods that generate the appropriate field-

232   oriented control voltages and add them to the queue of motor phase timings.

233   **Data collection**: Characterization data is stored in a custom 4x128 ring buffer

234   `motor_characterize_data` (which is modeled on the existing ODrive `oscilloscope` buffer). It is

235   defined globally so that it can be accessed by both `axis.cpp` (for data writing) and `communication.cpp`

236   (for user-accessibility). On each loop, `run_motor_characterize_input()` calls custom method

237   `record_motor_characterize_data()` with the latest timestep and voltage command, and

238   `record_motor_characterize_data()` writes them to `motor_characterize_data` along with the latest

239   encoder estimates of motor position and velocity.

240   **Data accessibility**: `motor_characterize_data` can be accessed by index in `odrivetool` using a set of

241   four get functions (`get_motor_characterize_data_timestep()`,

242   `get_motor_characterize_data_voltage()`, etc.) because these functions were defined in

243   `odrive_main.h` and added to the communication protocol in `odrive-interface.yaml`. At any given

244   time, the "latest recorded observation" can be accessed by calling the get functions at custom index

245   `motor_characterize_data_pos` (which is also user-accessible via a get function).

246   <u>Software – User Host Computer Python Code</u>

247   The ODrive itself has insufficient memory for large data files, so the USB user interface `odrivetool` was

248   edited to add a new method `run_motor_characterize_input()` that runs the input, pulls

249   characterization data from the firmware buffer, and exports it to CSV. Taking arguments odrv (ODrive

250   object, default `odrv0`), axis (0 or 1), and export directory (string), `run_motor_characterize_input()`

251   sets the `requested_state` of the specified axis to `AXIS_STATE_MOTOR_CHARACTERIZE_INPUT` (triggering

252   the state machine to begin the test input), and then continuously submits pull requests for the "latest

253   recorded" sample values of `motor_characterize_data` until the input ends.

254   Note that `odrivetool` can only be used when connecting over USB. This has drawbacks, since

255   the connection is slow, but `run_motor_characterize_input()` has been designed to mitigate this as

256   much as possible by saving all data to an array and only writing to CSV once data collection is finished.

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

258   Data is formatted such that the CSV can be loaded directly into MATLAB, where it is plotted and

259   analyzed. Simple plot analysis can be used to identify the time constant $\frac{1}{a}$ (e.g. by finding the settling

260   time T_s and taking $a = \frac{4}{T_s}$), steady-state value SSV, and gain $K = SSV * a$, and construct a typical DC

261   motor plant function $G(s) = \frac{K}{s(s+a)}$.

262   # Results & Discussion

263   Firmware and software edits successfully achieved reasonably high-resolution data collection using the

264   specified test inputs. The user may edit `<axis>.input_config` in `odrivetool`, then run the selected

265   test input and export the data to a local directory using `run_motor_characterize_input()` with

    12

266   arguments ODrive object (e.g. default odrv0), axis number (0 or 1), and export directory (string). Data

267   files are formatted such that they can be loaded into MATLAB for plotting and analysis (Fig. 4).

268   Sample rate was limited by USB connection and ODrive communication protocol standards.

269   Standard ODrive firmware communication protocols do not allow for export of more than a single value

270   on one call, so `run_motor_characterize_input()` was designed to access the "last-recorded index"

271   `motor_characterize_data_pos` on each loop, then run four individual get functions

272   (`get_motor_characterize_data_timestep()`, `get_motor_characterize_data_voltage()`,

273   `get_motor_characterize_data_position()`, and `get_motor_characterize_data_velocity()`) for

274   that index. This guarantees that all four pieces of data are from the same timestep, but also means that

275   data recording can only capture "snapshots" of the recorded data.

276   By optimizing `run_motor_characterize_input()`, it was possible on this test system to achieve

277   an average sample rate of 200Hz (Fig. 5). Since firmware-side data is recorded at 8kHz, this means data

278   exporting is capturing approximately one out of every forty observations. This is due in part to a

279   fundamental limitation of ODrive communication, which is that the ODrive's communication protocols

280   only allow the user to access one value (e.g. a float) at a time; in order to observe time, voltage,

281   position, and velocity, the user must therefore make four separate get function calls. Future versions of

282   the ODrive may include a "subscribe" feature for high-speed USB export of a small number of 32-bit

283   values, but the time frame for these additions is not yet decided (O. Weigl, owner, ODrive Robotics,

284   personal communication, April 8, 2021). In the interim, sample rate for this application could potentially

285   be improved by reducing the types of data collected. For example, if the voltage input is known then

286   characterization should only require timestep and velocity data; the application could be altered to pull

287   only those two, or to instead record timestep and position and then derive velocity estimates from the

288   position data. Finally, there is also the simple hardware limitation of USB speed. The ODrive does allow
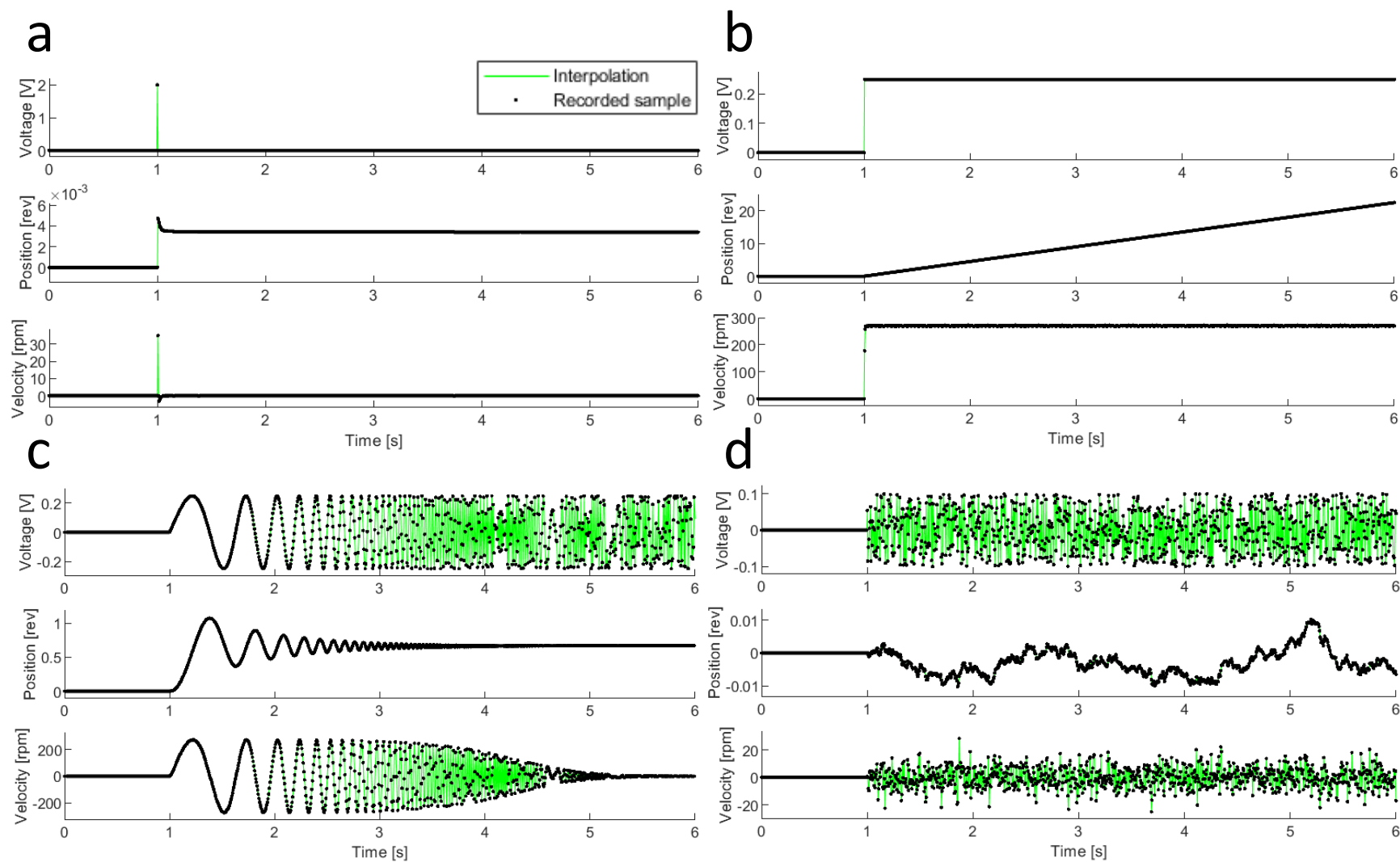
Figure 4. Actual motor data collected from the four newly-added test signals. Impulse (a), step (b), exponential chirp (c), and white noise (d) voltage inputs with encoder-recorded position and velocity responses.

290   for other communication protocols (UART, CAN, etc.), which would undoubtably be faster;

291   however, this would require an entirely new user interface, as `odrivetool` is limited to USB.
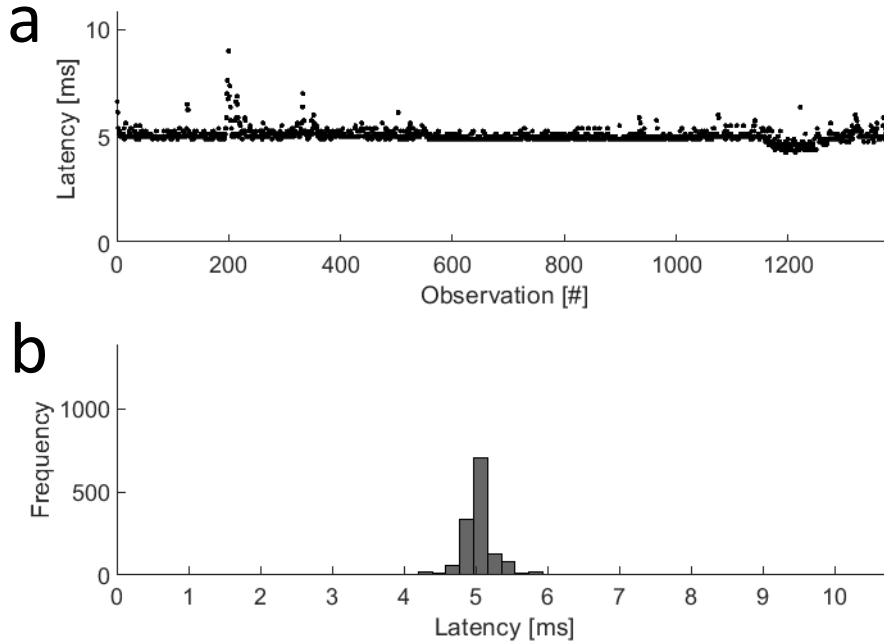

Figure 5. Typical Python-side latency for data collection (mean 5.0 ms) by sample (a) and histogram (b).

292   Despite these limitations, sampled data was sufficiently high-resolution to allow motor

293   characterization, since the bandwidth of the BLDC motor is well below kHz sampling rates. Based on

294   velocity step response for a quarter-volt step, characterization analysis produced a plant transfer

295   function $G(s) = \frac{1502}{s(s+333)}$ with time constant $\frac{1}{a} = \frac{1}{333.33} = 0.003\ s$; steady-state velocity $SSV =$

296    $4.5060\frac{rev}{s}$; 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

297   and MATLAB analysis code, see the *docs/motor_characterization_references* folder on the lab GitHub

298   [13].Using this transfer function, it should be possible to control the motor simply by deriving the

299   transfer function of the controller $C(s)$, finding $T(s) = G(s)C(s)$, and performing state space pole

300   placement with $T(s)$. Note, however, that finding $C(s)$ may be nontrivial depending on the controller in

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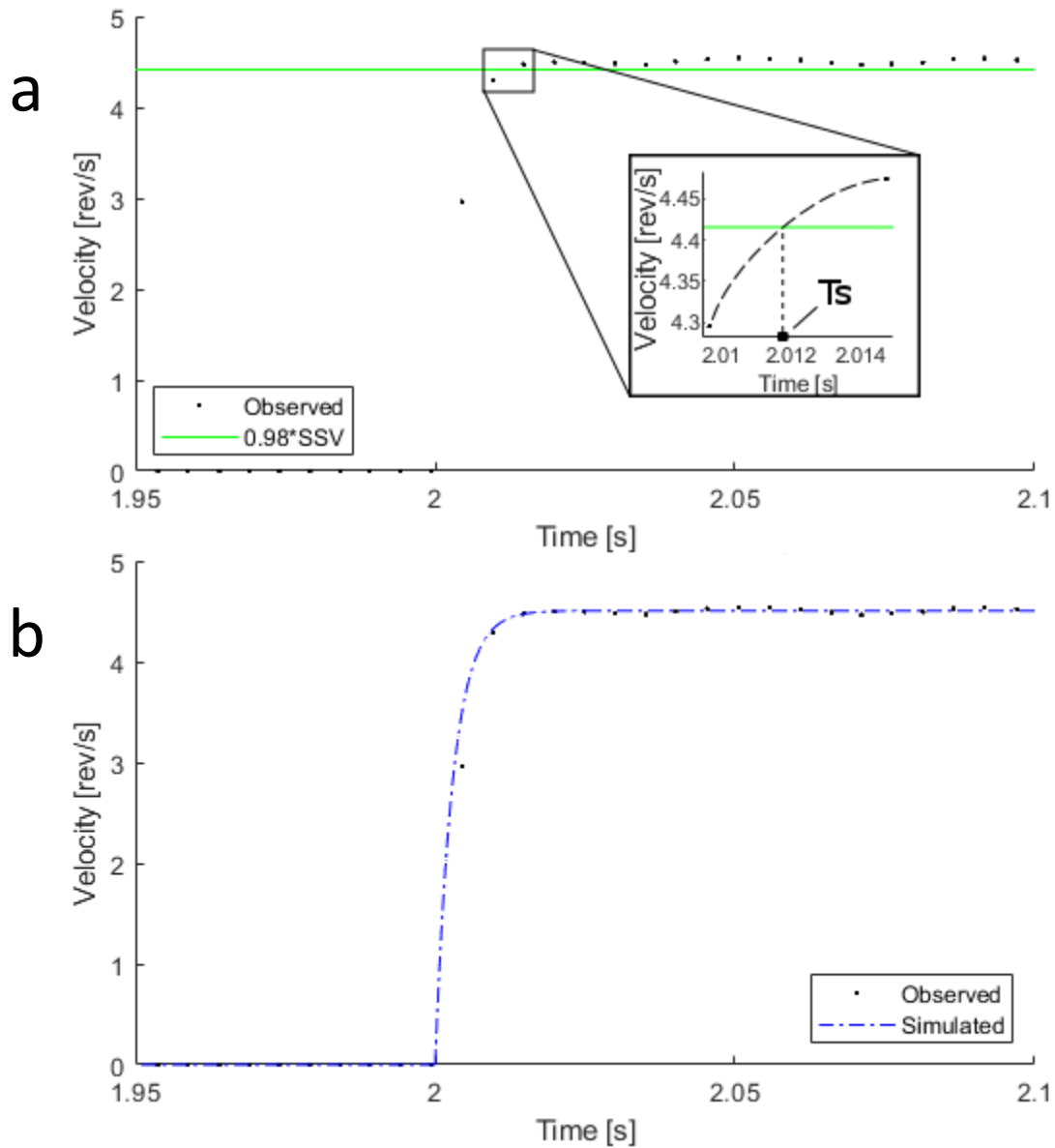
Figure 6. (a) Method to derive T_s from a step response. Plot the within-2% steady-state value $0.98 * SSV$; estimate where the step response crosses that line; and take that crossing time as T_s. From here, the transfer function can be estimated, in this case producing a simulated step response (b)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

302   control (see Appendix I for example analysis), of which the position and velocity loops are user-facing

303   and the current loop is not [5]. Alternatively, the user might choose to introduce their own controller;

304   this would require further editing of the firmware, but would substantially expand the possible control

305   options for more specialized applications.

16

Conclusions

This project successfully enabled motor characterization by adding test input and data export features to ODrive v0.5.1 firmware and software, though its sampling resolution was limited to 200Hz by software and hardware restrictions. Future work would benefit from any higher-speed additions to the ODrive's communication protocols; alternatively, there is room to optimize this project's additions by reducing the amount of recorded data to the minimum allowable for a given application. 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

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.

---

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

17

327 [3] C. Rollman, "What is 'Field Oriented Control' and what good is it?," *Copley Controls Corp.* pp. 1–

328 13, 2002.

329 [4] "Park, Inverse Park And Clarke, Inverse Clarke Transformations MSS Software Implementation

330 User Guide," *Microsemi Corporation*. Alieso Viejo, CA, pp. 5–9, 2013.

331 [5] "ODrive Documentation," *ODrive Robotics*, 2021. [Online]. Available: docs.odriverobotics.com.

332 [Accessed: 01-Apr-2021].

333 [6] "Motion System Tuning." Rockwell Automation, Milwaukee, WI, pp. 22–23, 2020.

334 [7] "ODrive Robotics GitHub," *ODrive Robotics*. [Online]. Available: github.com/odriverobotics.

335 [Accessed: 01-Apr-2021].

336 [8] "ODrive Firmware Developer Guide," *ODrive Robotics*. [Online]. Available:

337 docs.odriverobotics.com/developer-guide. [Accessed: 01-Apr-2021].

338 [9] "ODrive encoder guide," *ODrive Robotics*. [Online]. Available:

339 https://docs.google.com/spreadsheets/d/1OBDwYrBb5zUPZLrhL98ezZbg94tUsZcdTuwiVNgVqpU

340 /edit#gid=0. [Accessed: 01-Apr-2021].

341 [10] "ODrive motor guide," *ODrive Robotics*. [Online]. Available:

342 https://docs.google.com/spreadsheets/d/12vzz7XVEK6YNIOqH0jAz51F5VUpc-

343 lJEs3mmkWP1H4Y/edit#gid=0. [Accessed: 01-Apr-2021].

344 [11] R. Smoot, "What's the Difference Between an Incremental Encoder's PPR, CPR, and LPR?," *CUI*

345 *Insights: Motion*, 2021. [Online]. Available: cuidevices.com/blog/what-is-encoder-ppr-cpr-and-

346 lpr. [Accessed: 01-Apr-2021].

347 [12] E. Goldberg, "ODrive Setup Walkthrough." MRD Lab, University of Minnesota Department of

18

348   Mechanical Engineering, Minneapolis, MN, 2021.

349   [13]   "MRD Lab GitHub," *Medical Robotics and Devices Lab*. [Online]. Available: github.com/labmrd.

350   [Accessed: 01-Apr-2021].

351   [14]   E. Goldberg, "VS Code for ODrive." MRD Lab, University of Minnesota Department of Mechanical

352   Engineering, Minneapolis, MN, 2021.

353

354   **Appendix I. ODrive Controller Simplification**

355   In order to optimize controller gains for a given motor transfer function, it is necessary to know the

356   transfer function for the controller. Standard ODrive control uses a series of P and PI loops for position

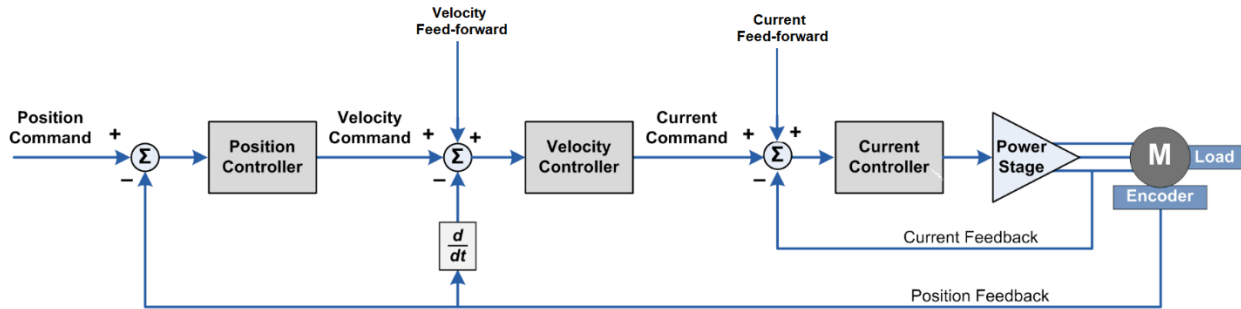357   control [5], of which only the position and velocity controller gains are user-settable.

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

358   Making the assumption that feed-forward terms are zero and that the current control loop can

359   be represented as a simple PI transfer function $I(s)$, we may use block diagram simplification methods

360   (Fig. A2) to derive an overall controller transfer function:

361
$$C(s) = \frac{\dfrac{P(s)I(s)V(s)}{1 + I(s)V(s)s}}{1 + \dfrac{P(s)I(s)V(s)}{1 + I(s)V(s)s}}$$

362

363
$$= \frac{P(s)I(s)V(s)}{1 + I(s)V(s)s + P(s)I(s)V(s)}$$

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

365   K_p, K_v, and K_{vi} may be defined by the user by setting `<odrive>.controller.config` members

366   `pos_gain`, `vel_gain`, and `vel_integrator_gain`, respectively. The current control loop, on the other

367   hand, is part of the Motor class, where K_i and K_{ii} correspond to the `p_gain` and `i_gain` members of

368   `<odrive>.motor.current_control`. `p_gain` and `i_gain` are not user-configured; they are instead set
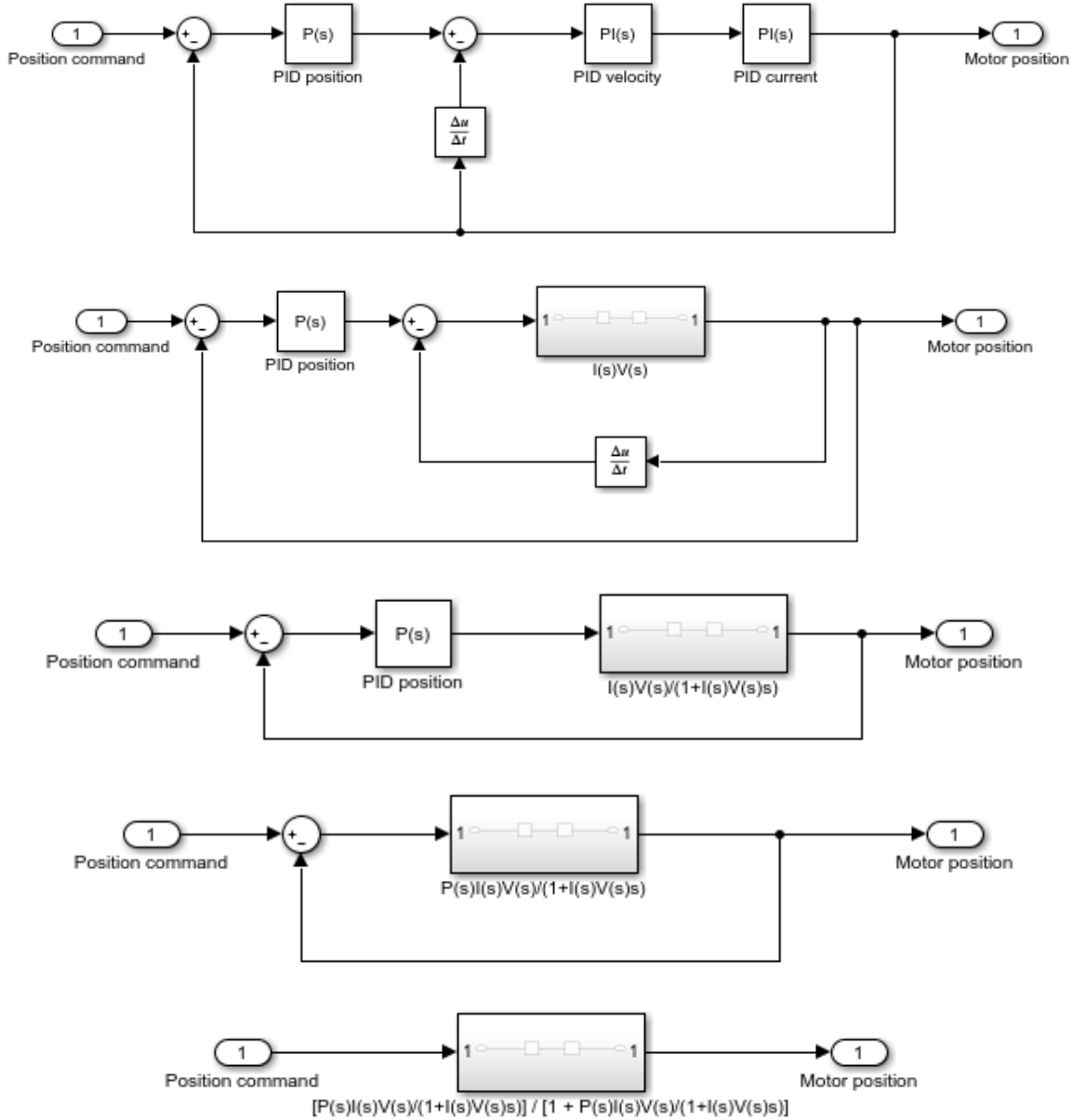
20

Figure A2. Block diagram simplification of ODrive controller.

369   automatically upon motor calibration as a function of the configured current control bandwidth and the

370   measured motor phase resistance and phase inductance:

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

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

373   If phase inductance and phase resistance are reasonably constant, it should be possible to treat K_i and

374   K_{ii} as observable constants for the purposes of control design.

21

375 **Summary of Contributions**

376 A list of all modifications made to ODrive v0.5.1. All edits are labeled with an "ERG" comment, so edits
377 may easily be viewed in the code [13] by searching "ERG".

378 Firmware: Input commands and data collection

379   I.   Test input
380   a. axis.cpp and axis.hpp
381   i. Added input configuration to Axis class
382   1. Added struct definition InputConfig_t
383   2. Added input_config argument (of type InputConfig_t) to Axis class
384   3. Add input_config_ field (of type InputConfig_t)
385   ii. Added new state machine behavior to run test input
386   1. Edited run_state_machine_loop() to include behavior for
387   AXIS_STATE_MOTOR_CHARACTERIZE_INPUT
388   2. Added method run_motor_characterize_input() (120 lines)
389   b. main.cpp
390   i. Added array input_configs
391   ii. Added input_configs to save_configuration() and load_configuration()
392   iii. Added input_config argument to axes initialization
393   II.   Data recording and user-accessibility
394   a. odrive_main.h
395   i. Added ring buffer motor_characterize_data, of size
396   MOTORCHARACTERIZEDATA_SIZE, with index tracker
397   motor_characterize_data_pos
398   ii. Added six get_motor_characterize_data_XXX() functions for buffer size, "latest
399   observation" index, and by-index timestep, voltage, position, and velocity
400   b. communication.cpp
401   i. Added initialization for ring buffer
402   c. odrive-interface.yaml
403   i. Added new AxisState MotorCharacterizeInput
404   ii. Added enum InputType
405   iii. Added input_config
406   iv. Added the six get functions
407   d. axis.cpp and axis.hpp
408   i. Added method record_motor_characterize_data() (8 lines), which writes data to
409   motor_characterize_data at index motor_character_data_pos when called as
410   part of run_motor_characterize_input()

411 Python: User interface, data retrieval and export

412   I.   enums.py - added AXIS_STATE_MOTOR_CHARACTERIZE_INPUT
413   II.   utils.py - added method run_motor_characterize_input() (88 lines)
414   III.   shell.py - added run_motor_characterize_input() to launch_shell()