# Balancing and Locomotion of an Inverted Pendulum

Duncan Mazza
Engineering Systems Analysis
Olin College of Engineering

Nathan Weil
Engineering Systems Analysis
Olin College of Engineering

***Abstract*—The goal of our project was to balance an two-wheeled robot while adding additional capability. The robot in question was Pololu Balboa modified such that it had a significantly higher center of mass. Using template self-balancing code provided by our professors, we performed characterization of the robot's motors and determined the effective length of the robot as a pendulum. We then adjusted the poles of the control system to produce control parameters that enabled the robot to balance upright with as little oscillatory movement as possible. After reaching this state, we extended the capability of the robot to respond to a pilot via a remote control. We were very successful with this, as our robot was very responsive to both throttle and turning commands while remaining upright indefinitely under nominal conditions.**

## I. Control Algorithm

### A. Static Balancing

The prescribed control algorithm for the robot is depicted in fig. 1 where:

- $d(t)$ is the signal of disturbances to the system
- $\theta(t)$ is the signal representing the angle of the pendulum (measured relative to vertical)
- System blocks $K_p$ and $K_i$ comprise a proportional-integral (PI) controller $K(s)$ that removes the error in the $\theta(t)$ signal; because the desired angle is 0°, $\theta(t)$ is itself the error feeding into the $K(s)$ controller
- $M(s)$ is the motor transfer function defined as

$$M(s) = \frac{ab}{s+a} \tag{1}$$

- $v_{d_{\neg w}}(t)$ is the desired velocity of the motor the wheel opposite of wheel $w$
- $v_{c_{\neg w}}(t)$ is the control velocity signal for the motor of the wheel opposite of wheel $w$
- System blocks $J_p$ and $J_i$ comprise a PI controller $J(s)$ that removes the error in velocity of the motors (which is ideally 0m/s)
- $\frac{C_i}{s^2}$ is the integral part of a PI controller $C(s)$ that removes the error in accumulated displacement of the motors (which is ideally 0m); the proportional part of the PI controller is not present because it would be redundant with $\frac{J_i}{s}$.
- $v_w(t)$ is the actual speed of wheel $w$ in m/s.
- $H_{v\theta}(s)$ is the transfer function relating the velocity of the base to the angle of the pendulum; it is defined as

$$H_{v\theta}(s) = \frac{s}{g - ls^2} \tag{2}$$

where $g$ is the acceleration due to Earth's gravity (9.85m/s) and $l$ is the effective length of the pendulum (see §II-A for calculation of $l$).

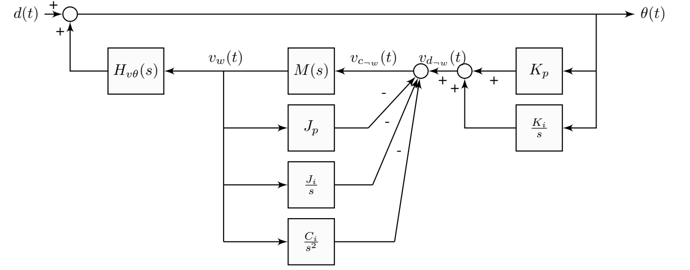


Fig. 1. Prescribed control diagram for the inverted pendulum (no modifications).

A difference between the provided control diagram and the one depicted in fig. 1 is the use of $w$ and $\neg w$ subscripts to indicate the nuance in which measurements and signals correspond to which wheel. The so-called criss-crossing of displacement measurements present in the provided control code is denoted by these subscripts. Because we implement turning capabilities in the following section, we thought it important to introduce these nuances in the diagram. Not captured by the control diagram is the fact that the transfer function $M(s)$ doesn't actually convert a signal from wheel $\neg w$ to $w$, and $H_{v\theta}(s)$ considers only the velocity of the base of the robot, not individual wheels; we will consider these disparities negligible for the sake of clarity in subsequent sections how turning is implemented.

### B. Modifications to Control System for Locomotion

To enable locomotion, we incorporated a displacement offset signal into the motor controlling part of the control system. Sepcifically, we added the per-wheel displacement to the integral term of the $J(s)$ controller and added the average of the wheels' displacement to the accumulated distance term of the $C(s)$ PI controller; see fig. 2 for context in how they are incorporated into the control algorithm. The per-wheel signal, $x_{r_w}(t)$ (note that the subscript $r$ is referring to remote control), is defined at time $t_i$ for each wheel $w$ as:

$$x_{r_w}(t_i) = \int_0^{t_i} v_{r_w}(t)dt \tag{3}$$

(because $v_r(t)$ is a discrete valued function in implementation, the integral is evaluated using a right Riemann sum).

Computing average of the wheels' displacement, $x_{r_{avg}}(t_i)$, is as follows:

$$x_{r_{avg}}(t) = \frac{x_{r_L}(t) + x_{r_R}(t)}{2} \quad (4)$$

where subscripts $L$ and $R$ correspond to the left and right wheels.

The signal $v_r(t)$ is defined independently for each wheel $w$ at time $t_i$ by:

$$v_{r_w}(t_i) = \begin{cases} T_{th} v_{r_{th}}(t) - T_{tu} v_{r_{tu}}(t) & \text{if } w = L \\ T_{th} v_{r_{th}}(t) + T_{tu} v_{r_{tu}}(t) & \text{if } w = R \end{cases} \quad (5)$$

where $v_{r_{th}}(t)$ and $v_{r_{tu}}(t)$ are the throttle and per-wheel control signals received from the remote control respectively, and $T_{th}$ and $T_{tu}$ are the parameters that scale the remote control's throttle and turning signals respectively to values reasonable for the motor controlling API. With trial-and-error tweaking, we found that $T_{tu} = 0.2$ and $T_{th} = 0.35$ worked well. The difference in sign of the $T_{tu} v_{r_{tu}}(t)$ term in eq. (5) accounts for the wheels spinning in opposite directions according to the direction the pilot tells the robot to turn. Code for the functionality described in this section can be found in §V-A.

An important nuance to note in fig. 2 is that the signs of $x_{r_w}(t)$ and $x_{r_{avg}}(t)$ are negative in their respective summation nodes; an intuitive way of understanding this is that it makes the control system think that the displacement of the robot is behind where it should be.

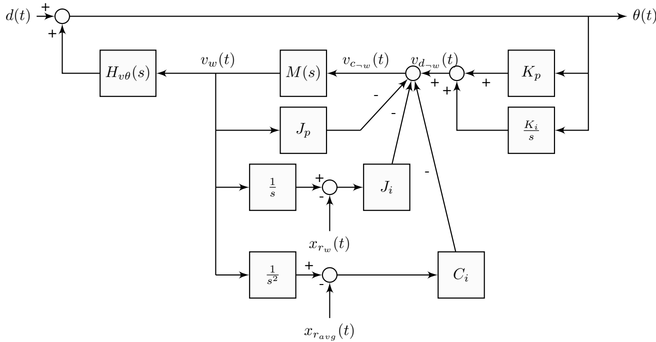


Fig. 2. Modified control system enabling input from the remote control to drive the robot.

### C. Integrating Remote Control in Hardware and Firmware

To integrate remote control capability into the hardware and firmware, we began by installing a small radio receiver on the robot. Digital pins 2 and 3 on the PCB were used as pulse width modulation (PWM) inputs. These pins were chosen because of their pin-interrupt capabilities, enabling a fast response and smooth PWM data input. The signal communicated over pin 3 corresponds to the throttle control, and the signal communicated over pin 2 corresponds to the turning control. To read a PWM signal, the following was performed for each pin individually:

1) When a signal pulse switches to high, the interrupt triggers a function that records the start time of the pulse
2) The main loop function then continues until the signal pulse returns to low, when a second interrupt function compares the pulse start time to the end time.
3) This change in time is recorded as the pulse width. We then shifted this PWM signal from its input range of [1000 $\mu$s, 2000 $\mu$s] to be zero-centered; this value corresponds to the respective $v_{r_{th}}(t)$ or $v_{r_{tu}}(t)$ signal.

The code implementing this capability can be found in §V-A.

## II. Calculating Hardware-Intrinsic Parameters

### A. Calculating Effective Length

In order to calculate the effective length of the inverted pendulum, we began by doing a swing test of our robot. We removed the wheels and held the robot upside-down by the axles. The on-board Arduino ran provided code that output gyro angle to the serial monitor on a laptop. We pushed the robot to an angle of approximately 30 degrees, and let it swing freely for three seconds. The data for this experiment is shown in fig. 3.

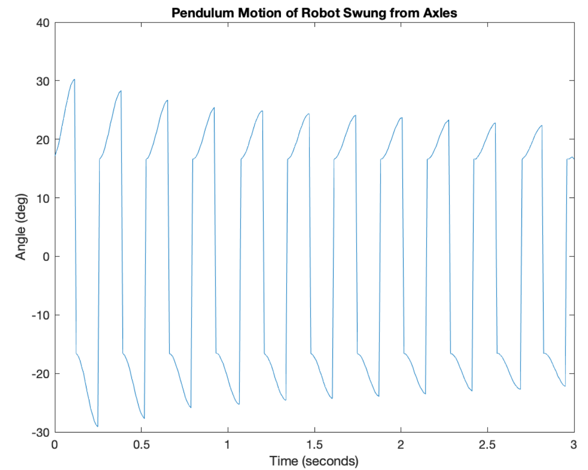


Fig. 3. Data from our natural frequency measurement used to calculate the effective length of the inverted pendulum.

To find the natural frequency $\omega_n$ of the robot as a pendulum, we wrote a MATLAB function (§V-B3) that takes the Fourier transform of the recorded data. The function returns the frequency corresponding with the largest value in the magnitude of the Fourier transform. We found this value to be $\omega_n = 4.77$ rad/sec for our robot.

We calculated the effective length $l$ of the robot as a pendulum by using

$$\frac{1}{\omega_n} = 2\pi\sqrt{\frac{l}{g}} \quad (6)$$

where $g$ is acceleration due to gravity. Using the value $\omega_n$ found above, we solved for $l$ and found a value of $l = 0.431$m.

### B. Calculating Parameters for Motor Transfer Function

Recall the motor transfer function defined in eq. (1). The step response of this transfer function with zero initial conditions to the maximum control velocity signal, 300 (a unitless parameter mapping to the maximum motor speed), is as follows:

$$v_{step}(t) = 300b(1 - e^{-at}) \tag{7}$$

To calculate the parameters $a$ and $b$, we performed the following experiment while collecting data on the wheel speed of each of the robot's wheels: The robot was held upright and stationary by one of us until both of the motors received the $v_c(t)$ control signal of 300 for 3 seconds (an arbitrary amount of time allowing the robot to reach its steady state velocity); as the robot accelerated, the person holding the robot upright followed the robot and attempted to keep it as upright as possible. The measured velocities for each wheel were nearly identical, and only the average of the wheel velocities was considered for calibration; this average wheel velocity can be seen in fig. 4. Using MATLAB's curve fitting tool to fit a curve of the form of $v_{step}(t)$ yielded two adequate values for $a$ and $b$ (denoted in fig. 4).

While the human intervention in this test makes it hard to reproduce exactly and introduces some error, the high performance of our robot's balancing and driving capabilities suggest that it served its purpose well.

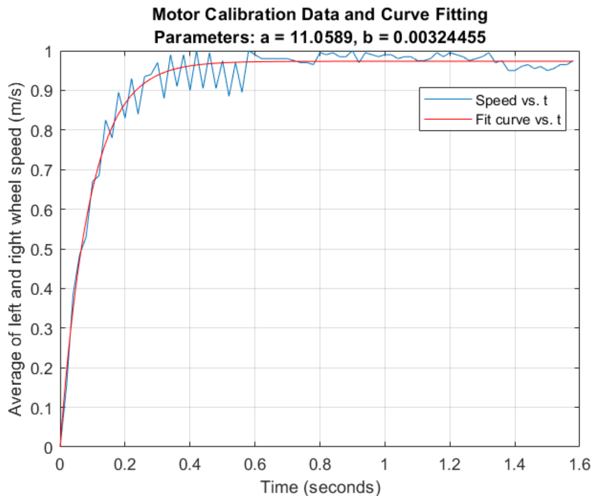


Fig. 4. Calibration data (blue) and fit curve (red) used to generate values for the parameters $a$ and $b$ used to define $M(s)$.

## III. Challenges

### A. Stationary Balancing

After determining $l$ the parameters for $MP(s)$, we solved for the control parameters using the default[1] system poles of:

- $p_1 = -0.5 + w_n i$
- $p_2 = p_1^*$
- $p_3 = -1$
- $p_4 = p_3^*$
- $p_5 = -14$

We used the default poles to calculate system function coefficients using code shown in §V-B2. When we input the default system function coefficients into the control code, the robot oscillated somewhat aggressively and and drifted in position over time. In order to make the robot more stable and prevent drifting, we modified the poles.

- $p_1 = -4 + 2\omega_n i$
- $p_2 = p_1^*$
- $p_3 = -2$
- $p_4 = p_3^*$
- $p_5 = -14$

The rationale for the above changes can be summarized with the following:

- The real component of all poles except $p_5$ became more negative to create a stronger decay of errors resulting from disturbances
- The imaginary components of $|p_1|$ and $|p_2|$ increased to combat the undesirable oscillations introduced by the original parameters.

Note that these alterations cause a change in the dominant response from the oscillating and exponentially decaying response to a purely exponentially decaying response, as the right most pole is repeating. When we input the new system function coefficients calculated from the altered poles, the robot was more stable in both oscillation and position, giving us a resilient platform for implementing remote control.

### B. Locomotion

The main challenge we encountered was to make our robot controllable via remote control. Specifically, our goal was for a person to pilot the robot using a hobby-grade radio transmitter, resulting in the robot moving effortlessly through its environment while being responsive to the input. We tried multiple methods of controlling the robot's motion before landing on a final, relatively simple solution.

We began by tackling PWM inputs from the radio receiver. After researching ways of reading PWM signals, we decided to use the more challenging interrupt pin method due to its reported smoothness and reliability. It proved easier than expected to implement, and provided smooth data with no noticeable delay on the control loop.

The largest challenge on our way to remote control was integrating the PWM signal from the receiver into the existing control loop for the robot. We first attempted control by adding the mapped input signal value directly to the motor output for each wheel. Though this intuitively seemed like it should work, the resulting movement was not what we expected. Because the integral term in the main control loop 'pulls' the robot's position back towards zero, directly manipulating the velocity controlled the position of the robot on the ground. We found this to be a fascinating effect of the control system design.

[1]The poles that follow are unchanged from the provided poles, except for poles $p_1$ and $p_2$

Although it was technically controlled remotely at that point, we had not met goal of effortless and responsive control.

After considering a few other options such as adding an offset to angle, we realized that the position control issues we faced stemmed from the distance offset we were creating as we pushed the robot away from its ”home” position. To solve this, added the control input to the wheel position measurements before the data enters the main control loop. This caused the controller input to change the position like before. However, the control was much more responsive and smooth, helping the robot's stability. To switch from positional control to velocity control, we integrated the input value and subtracted it from the wheel positions, defining our own constant offset as described above.

## IV. Results

We were very successful with achieving our goals:

- The robot remained upright unless it experienced a significant disturbance
- With no remote control signal, the robot remained in place
- Under remote control, the robot was agile and responded quickly to inputs

A video of our robot being demonstrated in class can be found at this link: https://youtu.be/ufF1Aeu0s3w.

## V. Code

### A. Control Code

Below is initialization and description of the required variables for the remote control inputs.

```
1  // volatile variables for pwm inputs from ...
       receiver
2  volatile int steer_value = 0; // Steering ...
       PWM value
3  volatile int throttle_value = 0; // Throttle ...
       PWM value
4
5  volatile int prev_timeS = 0; // PWM interupt ...
       pin timer setup
6  volatile int prev_timeT = 0;
7
8  float steering = 0.;    // mapped steering value
9  float throttle = 0.;    // mapped throttle value
10 float l_input = 0.;     // left input value
11 float r_input = 0.;     // right input value
12
13 float s_coeff = .2;     // steering map ...
       factor (controls steering sensitivity)
14 float t_coeff = .35;    // throttle map ...
       factor (controls throttle sensitivity)
15
16 float steer_accum = 0.; // integral of ...
       steering input
17 float throttle_accum = 0.; // integral of ...
       throttle input
```

Below is an exceprt of the code that:

- Maps the inputs from the remote control
- Calculates the displacement accumulated by the remote control signal and adds it to the measured displacement of the wheels
- Calculates $v_{c_L}$ and $v_{c_R}$:

```
1  // Remap steering and throttle values to 0 ...
       centered and max magnitude of ...
       sensitivity inputs
2  steering = mapfloat((float)steer_value, ...
       1000., 2000., -s_coeff, s_coeff);
3  throttle = mapfloat((float)throttle_value, ...
       1000., 2000., -t_coeff, t_coeff);
4
5  // Integrates mapped steering and throttle ...
       inputs
6  steer_accum += steering*0.01;
7  throttle_accum += throttle*0.01;
8
9  v_d = Kp*(angle_rad) + Ki*(angle_rad_accum); ...
       // this is the desired velocity from the ...
       angle controller
10
11 // The next two lines implement the feedback ...
       controller for the motor. Two separate ...
       velocities are calculated.
12 // We use a trick here by criss-crossing the ...
       distance from left to right and
13 // right to left. This helps ensure that the ...
       Left and Right motors are balanced
14 v_c_R = v_d - Jp*measured_speedR - ...
       Ji*distLeft_m - dist_accum*Ci;
15 v_c_L = v_d - Jp*measured_speedL - ...
       Ji*distRight_m - dist_accum*Ci;
```

Below is the code that adds the `steer_accum` and `throttle_accum` terms to the integrated distance terms for each wheel:

```
1  distLeft_m = ((float) distanceLeft) / ...
       ((float) G_RATIO) / 12.0 * 80.0 / 1000.0 ...
       * 3.14159 - steer_accum - throttle_accum;
2  distRight_m = ((float) distanceRight) / ...
       ((float) G_RATIO) / 12.0 * 80.0 / 1000.0 ...
       * 3.14159 + steer_accum - throttle_accum;
3  dist_accum += (distLeft_m + distRight_m) * ...
       0.01 / 2.0;
```

Below are the functions which measure and record the PWM input from the radio receiver. The rising functions record the time code at which the voltage rises. The falling functions calculate the width of the pulse when the voltage falls.

```
1  void risingS() {
2   attachInterrupt(2, fallingS, FALLING);
3   prev_timeS = micros();
4  }
5
6  void risingT() {
7   attachInterrupt(3, fallingT, FALLING);
8   prev_timeT = micros();
9  }
10
11 void fallingS() {
12  attachInterrupt(2, risingS, RISING);
13  steer_value = micros()-prev_timeS;
14 }
15
16 void fallingT() {
17  attachInterrupt(3, risingT, RISING);
18  throttle_value = micros()-prev_timeT;
19 }
```

## B. Analysis Code

*1) Code for Calculating Control Parameters:* Running the following code solves for the values for the control parameters $Kp$, $K_i$, $J_p$, $J_i$, and $C_i$ using the transfer function of the control system and the desired poles:

```
% add paths for functions not in directory
addpath("Natural Frequency Calculation\Data ...
    and Analysis\");
addpath("Motor Transfer Function ...
    Calibration\Data and Analysis\");

% acceleration due to gravity
g = 9.85;

% data for calculating natural frequency
frequency_test_file = "Natural Frequency ...
    Calculation\Data and ...
    Analysis\swing_data.mat";
load(frequency_test_file)

% calculating effective length of pendulum
fs = 1/0.05; % sample frequency
wn = find_strongest_freq(swing_data, fs);
l = g / wn.^2;

% data for calculating motor parameters
motor_test_file = "Motor Transfer Function ...
    Calibration\Data and Analysis\test1.mat";

% motor parameters
[a, b] = findAandB(motor_test_file, false);

% system poles
p1 = -4 + wn*2i;
p2 = conj(p1);
p3 = -2;
p4 = conj(p3);
p5 = -14;

[Jp, Ji, Kp, Ki, Ci] = findControlParams(wn, ...
    a, b, l, p1, p2, p3, p4, p5);
```

The output for this script is printed into the MATLAB console in the following format:

```
#define Jp 416.408
#define Ji -8415.92
#define Kp 7679.4
#define Ki 40041.9
#define Ci -7341.95
```

These compile directives can be supplanted into the control code (§V-A) to modify the robot's behavior. This code has three dependencies for user-defined functions not provided in the assignment shown; they are shown in the following three sections.

*2) Code for Solving for the Control Parameters Given Poles:* Below is the code that, given all of the parameters intrinsic to the robot's hardware and 5 desired poles (`p1`, `p2`, `p3`, `p4`, and `p5`), returns the control system parameters:

```
function [Jp, Ji, Kp, Ki, Ci] = ...
    findControlParams(wn, a, b, l, p1, p2, ...
    p3, p4, p5)
syms s Kp Ki Jp Ji Ci % define symbolic ...
    variables
g=9.85;
Hvtheta = -s/l/(s^2-g/l); % TF from velocity ...
    to angle of pendulum
K = Kp + Ki/s; % TF of the angle controller
J = Jp + Ji/s + Ci/s^2; % TF of the ...
    controller around the motor
M = a*b/(s+a); % TF of motor
Md = M/(1+M*J); % TF of motor + feedback ...
    controller around it
            % J is applied on the feedback path

% this is the total transfer function from ...
    disturbance d(t) to \theta(t)
Htot = 1/(1-Hvtheta*Md*K);

% this is the target characteristic polynomial
tgt_char_poly = ...
    (s-p1)*(s-p2)*(s-p3)*(s-p4)*(s-p5);

% get the denominator from Htot_subbed
[¬, d] = numden(Htot);

% find the coefficients of the denominator ...
    polynomial TF
coeffs_denom = coeffs(d, s);

% divide out the coefficient of the highest ...
    power term
coeffs_denom = coeffs(d, s)/(coeffs_denom(end));

% find coefficients of the target ...
    charecteristic polynomial
coeffs_tgt = coeffs(tgt_char_poly, s);

% solve the system of equations setting the ...
    coefficients of the
% polynomial in the target to the actual ...
    polynomials
solutions = solve(coeffs_denom == ...
    coeffs_tgt, Jp, Ji, Kp, Ki, Ci);

% display the solutions as double precision ...
    numbers
Jp = double(solutions.Jp);
Ji = double(solutions.Ji);
Kp = double(solutions.Kp);
Ki = double(solutions.Ki);
Ci = double(solutions.Ci);
fprintf("#define Jp %g\n#define Ji ...
    %g\n#define Kp %g\n#define Ki ...
    %g\n#define Ci %g\n", Jp, Ji, Kp, Ki, Ci);
impulse_response_from_sym_expression(subs(Htot))
```

*3) Code for Determining the Pendulum's Natural Frequency:* Below is the code that, when given a time series `x` sampled at frequency `Fs`, returns the radial frequency of the strongest frequency present in the signal:

```
function max_mag_freq = ...
    find_strongest_freq(x, Fs)
%Returns the frequency (in rad/sec) ...
    corresponding with the largest value in
%the magnitude of the FFT
%   x = input signal
%   Fs = sampling frequency in Hz
if (mod(length(x),2) ≠0)
    x = x(1:end-1);
end
mid_idx = floor(length(x)/2)+1;
freq_Hz_all = linspace(-Fs/2, ...
    Fs/2-Fs/length(x), length(x));
freq_Hz_right_handed = freq_Hz_all(mid_idx:end);
mag_FFT_all = 1/length(x)*fftshift(abs(fft(x)));
```

```
13 mag_FFT_right_handed = mag_FFT_all(mid_idx:end);
14 max_mag_pos = find(mag_FFT_right_handed == ...
       max(mag_FFT_right_handed));
15 if (length(max_mag_pos) > 1)
16     % in the case that there is a tie for the ...
           maximum magnitude, pick the
17     % first instance
18     max_mag_pos = max_mag_pos(1);
19 end
20 max_mag_freq = ...
       2*pi*freq_Hz_right_handed(max_mag_pos);
```

*4) Code for Curve-fitting Step Response Data:* Below is the code that fits a curve to motor velocity data from a step response run; the data is loaded from a struct located at path `data_file`:

```
1  function [a, b] = findAandB(data_file, ...
       plot_bool)
2  %Finds the a and b parameters of the motor ...
       speed/control transfer function
3      % Parameters:
4      %  data_file is the path to the file of ...
           the left and right wheel
5      %  velocity data
6      %  plot_bool is a boolean for whether the ...
           result of fitting is plotted
7      % The data_file can be generated by ...
           following these steps:
8      %  1) Copy the output from the serial ...
           monitor of the left and right
9      %  wheel speeds. Assuming it is formatted ...
           correctly, you can paste it
10     %  in as follows:
11     %     motor_vel_data = [paste here]
12     %  2) Then run the following commands to ...
           save the .mat file by filling
13     %  in the filename and the sample rate ...
           (in seconds) at which the data
14     %  was collected:
15     %     test_data = ...
           struct("motor_vel_data", ...
           motor_vel_data, ...
16     %         "sampling_rate", 0.02)
17     %     save 'filename.mat' test_data
18
19     load(data_file, 'test_data');
20     motor_vel_data_dims = ...
           size(test_data.motor_vel_data);
21     if motor_vel_data_dims(2) ≠2
22         error("Provided motor velocity data ...
               must be in the form of " + ...
23             "a nx2 matrix")
24     end
25
26     t = linspace(0, ...
           (length(test_data.motor_vel_data) - ...
           1) * ...
27         test_data.sampling_rate, ...
               length(test_data.motor_vel_data));
28     RandLAvg = (test_data.motor_vel_data(:, ...
           1) + ...
29         test_data.motor_vel_data(:, 2))./2;
30
31     [xData, yData] = prepareCurveData(t, ...
           RandLAvg');
32
33     % Set up fittype and options.
34     ft = fittype( '300*b*(1-exp(-a*x))', ...
           'independent', 'x', ...
35         'dependent', 'y' );
36     opts = fitoptions( 'Method', ...
           'NonlinearLeastSquares' );
37     opts.Display = 'Off';
38     opts.StartPoint = [0.901700858801764 ...
           0.184296530601589];
39
40     % Fit model to data.
41     [fitresult, ¬] = fit( xData, yData, ft, ...
           opts );
42     a = fitresult.a;
43     b = fitresult.b;
44
45     if plot_bool
46         fprintf("a: %g\n", a);
47         fprintf("b: %g\n", b);
48         % Plot fit with data.
49         figure('Name', 'Motor test parameter ...
               fitting');
50         h = plot(fitresult, xData, yData, '-');
51         legend(h, 'Speed vs. t', 'Fit curve ...
               vs. t', 'Location', ...
52             'NorthEast', 'Interpreter', 'none');
53         % Label axes
54         xlabel('Time (seconds)', ...
               'Interpreter', 'none');
55         ylabel('Average of left and right ...
               wheel speed (m/s)', ...
56             'Interpreter', 'none');
57         title(sprintf("Motor Calibration Data ...
               and Curve Fitting\nParameters: a = ...
               %g, b = %g", a, b))
58         grid on
59     end
60 end
```