12.444 Pset 2

Julian Leland

2016-12-10

**Objectives:** In this project, we leverage the signal-processing techniques taught in 12.444 Module 2 to support analysis of a dataset used in robot arm performance characterization in accordance with the ISO 9283 standard. The data analyzed here tracks the 3D position of the endpoint of a robot arm as it moves through a path in space. The data is collected using a Leica emScon laser tracker system, which can provide high-precision position data at reasonably high measurement frequencies (up to 10 Hz for limited durations).

However, for the particular ISO 9283 characterization being performed here, it is important to not just measure the robot’s position over time, but also be able to detect when the robot has stopped at various waypoints. The Leica and the robot’s control architecture are challenging to synchronize in any way that would allow coordination of measurements with robot movements, so we instead turn to post-run signal processing to determine a) when the robot has come to a full stop, and b) where in space it is.

The specific goals of this project are:

1. Write parsing code that reads through each dataset and detects/logs 1) times where all three signals (X, Y and Z) become stationary, and 2) times where all three signals are stationary at the defined "starting point" for the dataset.
2. Calculate basic statistics of measured data. My first goal is to return FFTs of the data measured during motion and during stopped phases, to get a sense of the frequency content of the noise inherent in my data. I expect to see some low-frequency (1-10 Hz) vibration from the system engine, but will be curious what else I can find.
3. Write code to create a randomly-generated dataset with selectable parameters (number of stopping points, amount of noise), which I can then use to experiment with robustness of processing code.

**Process:**  Initially, we start work with a single dataset, ‘ExampleISO9283Data\_Longer’. This dataset contains three channels corresponding to X, Y and Z position data, plus an index vector. While not shown clearly in the dataset, the data was collected at 3 Hz (visible in original dataset in text field).

After import and addition of a time vector, our first step is to plot our base dataset, so we can see what we are working with. We plot the three position measurement channels individually as a function of time, along with a 3D plot of XYZ position:

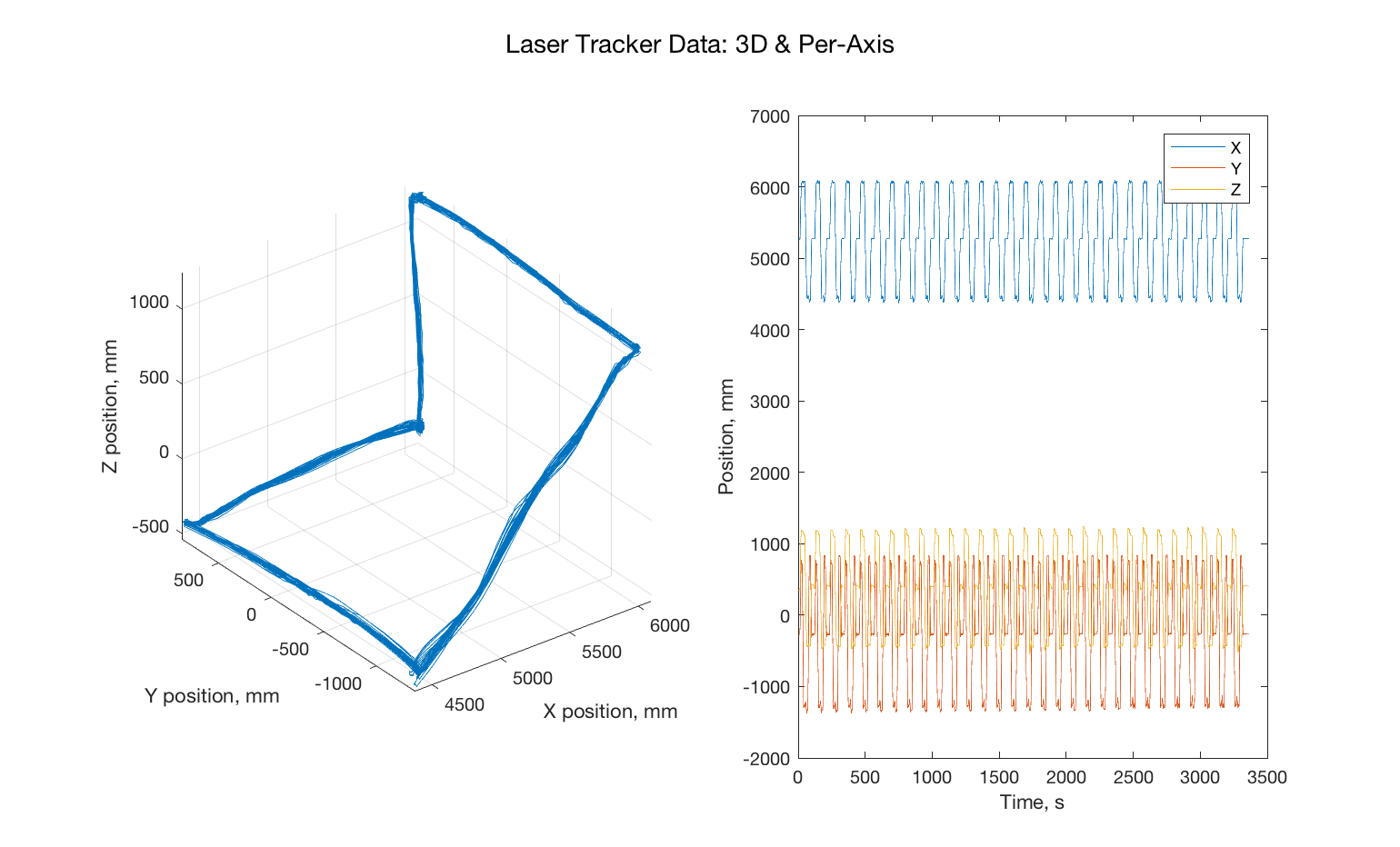


Fig. 1: Laser Tracker Data, 3D and Per-Axis

From these plots, we can verify that the data has accurately captured the trajectory the robot has transcribed in 3D, and there are no strange transcription errors (e.g, mis-sorted entries in the plot, which could show up as sudden jumps from one part of the trajectory to another). The per-axis plot also gives us a sense of how many complete cycles of the trajectory were completed, and of the relative DC offset of each signal (which is important for ensuring appropriate scaling later during frequency analysis.

Having examined the data spatially, we now examine it in the frequency domain. We first normalize each position channel around the mean value of that channel, to eliminate any DC offset in the signal from skewing our frequency spectrum amplitudes. We then take the FFT of our position data, extract the one-sided spectrum, and plot. We plot individual signals at frequency ranges from 0 to 0.5 Hz, 0.5 to 1 Hz, and 1 to 1.5 Hz.

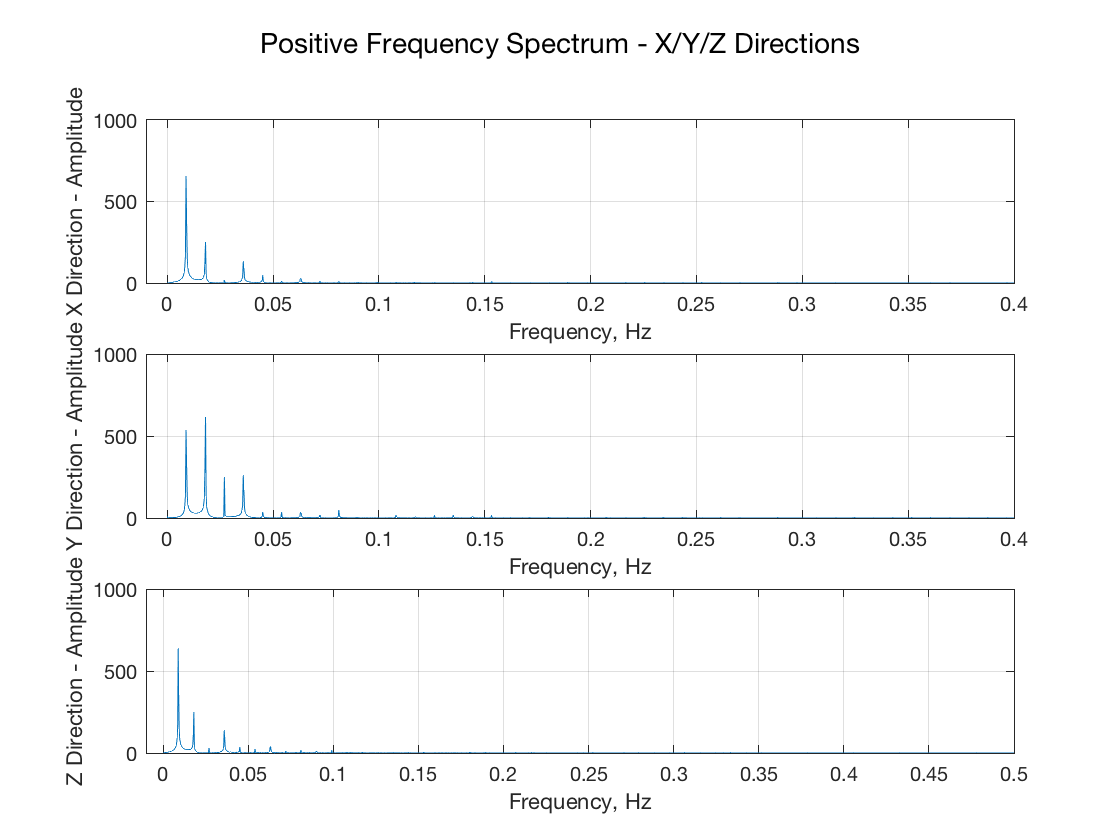


Figure 2: Frequency Spectrum, 0 to 0.5 Hz

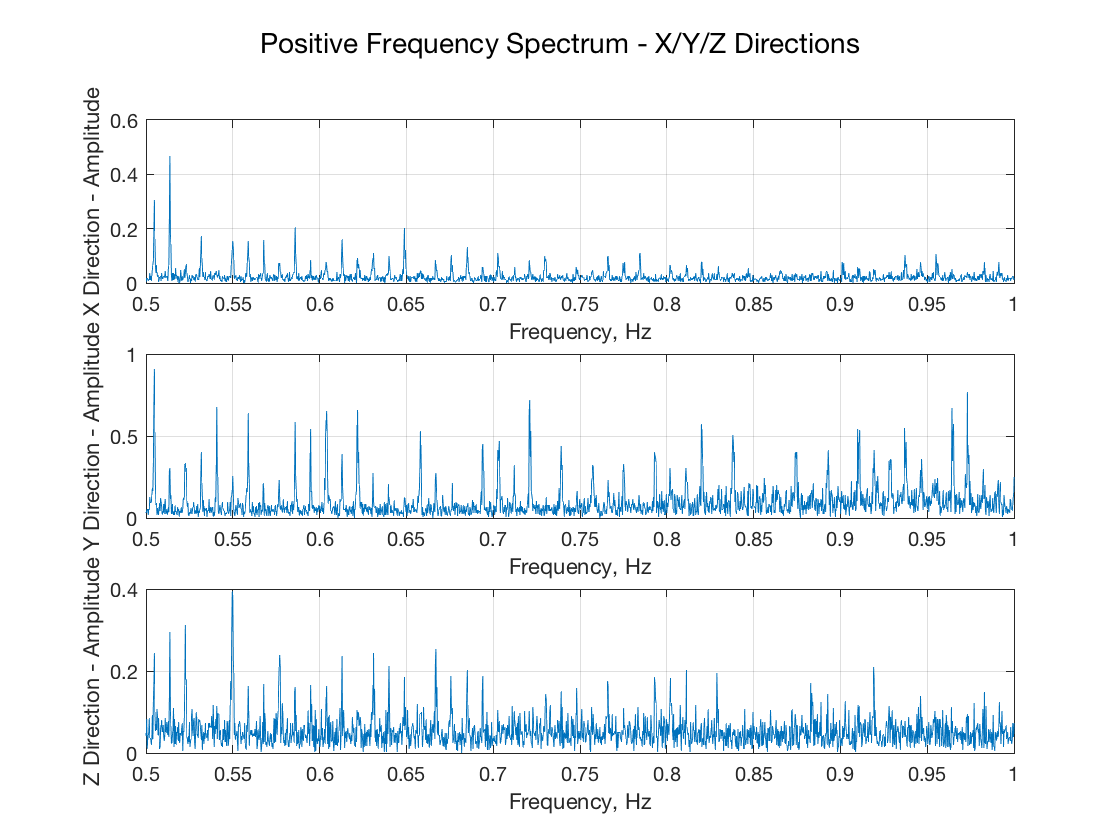


Figure 3: Frequency Spectrum, 0.5 to 1 Hz

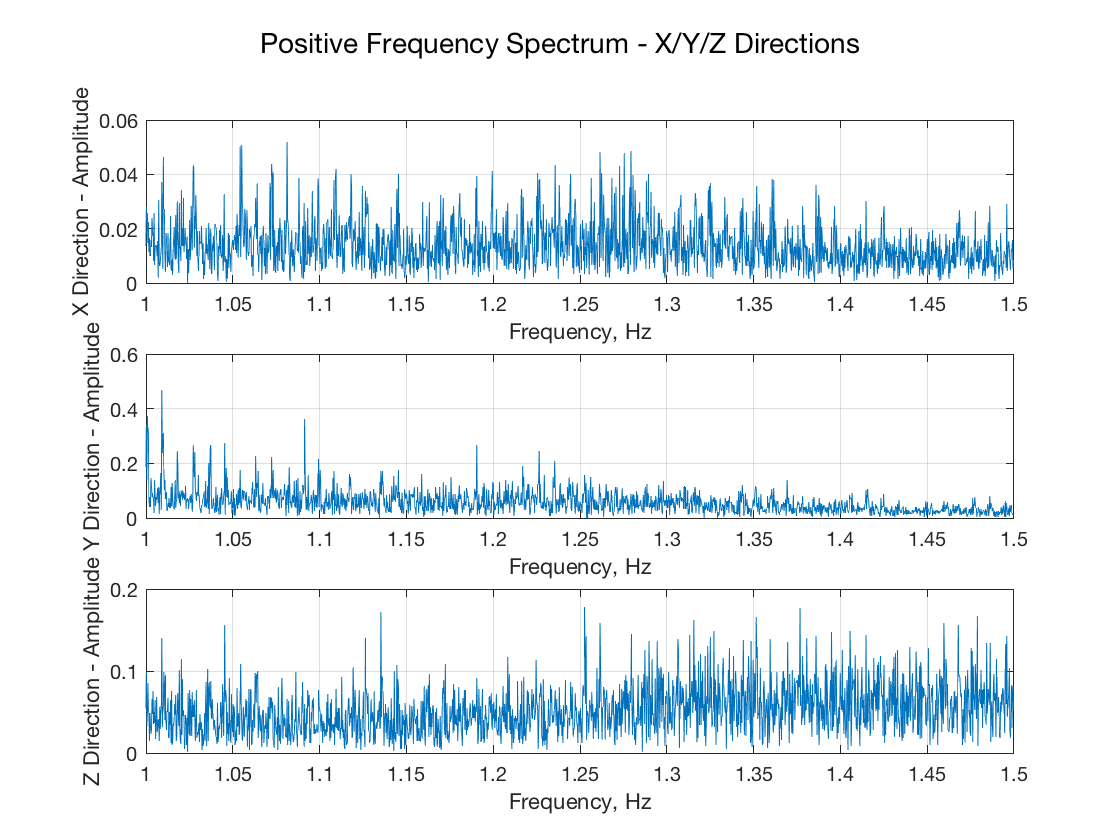


Figure 4: Frequency Spectrum, 1 to 1.5 Hz

As our measurement system only samples at 3 Hz, our ability to detect higher-frequency signals of interest – such as the oscillation of the robot’s boom, or the vibration introduced by the robot’s hydraulic motor is limited. However, we can identify a few points of interest in these plots:

* First, in the lowest range of the spectrum, we can see a number of distinct peaks, some of which overlap between channels.
  + The lowest peak occurs at 0.0089 Hz, or ~112 s, which corresponds quite closely to the time required to complete a single cycle of the pattern. While this peak should be seen regardless of pattern shape, it may be amplified by the simultaneous X/Y/Z movement of the system as it leaves/returns to its center position, which occurs once per cycle. This will be an interesting question to experiment with when we begin to create our own dummy paths to test this filtering software on.
  + As expected, the Y direction contains a number of higher-frequency peaks as well, corresponding to the greater amount of motion in this axis.
* At higher frequency ranges, while we cannot clearly identify specific peaks, the relative signal power in these ranges may be instructive. Previous testing with our robot has shown that the natural frequencies of arm oscillation range between roughly 0.6 and 2.5 Hz. One of the most pronounced modes observed is lateral oscillation of the boom, which is easily excited by any lateral movement of the arm, and which has been measured between 0.6 and 0.9 Hz depending on the degree of arm extension. Looking at the plot of Y-axis frequency in Fig. 3 above (the range at which we expect to see lateral boom oscillation), while there is no specific peak that we can identify as representing the lateral oscillation of the boom, the increased amplitude of signals across this range relative to the other two directions is quite interesting. While our robot’s controller is doing an adequate job of avoiding excitation of the boom’s lateral mode, the mode still exists.

With this information, we now turn to filtering. We first wish to low-pass filter our data, so that any filters that we use to detect when our robot has reached a steady state are not affected by noise. Our frequency analysis above has shown that we can confidently reject any signals slower than ~0.2 Hz. We start with a highly aggressive Butterworth filter, with a passband of 0.05 Hz and a stopband of 0.1 Hz. These criteria result in a 10th-order filter with a -3dB frequency of 0.054 Hz.

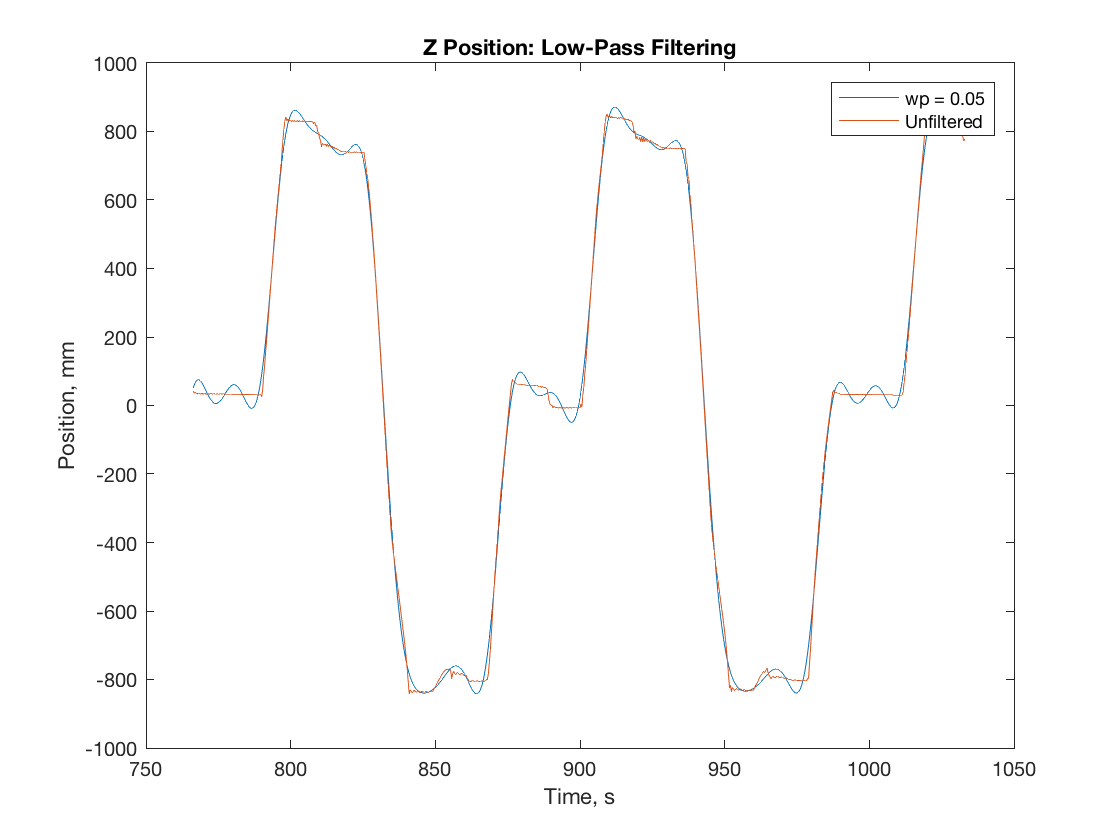


Figure 5: Low-pass filtered data, wp = 0.05 Hz

However, as Fig. 5 shows, this filter is too aggressive. This filtering has completely drowned out the small position steps that occur at the extreme ends of the trajectory, and as the trajectory passes through 0 at ~900s. We need a slightly less aggressive filter, which will not eliminate these structures. These structures are roughly 12 seconds in length, corresponding to a frequency of ~0.08 Hz, so we push our filter passband higher, to 0.1 Hz.

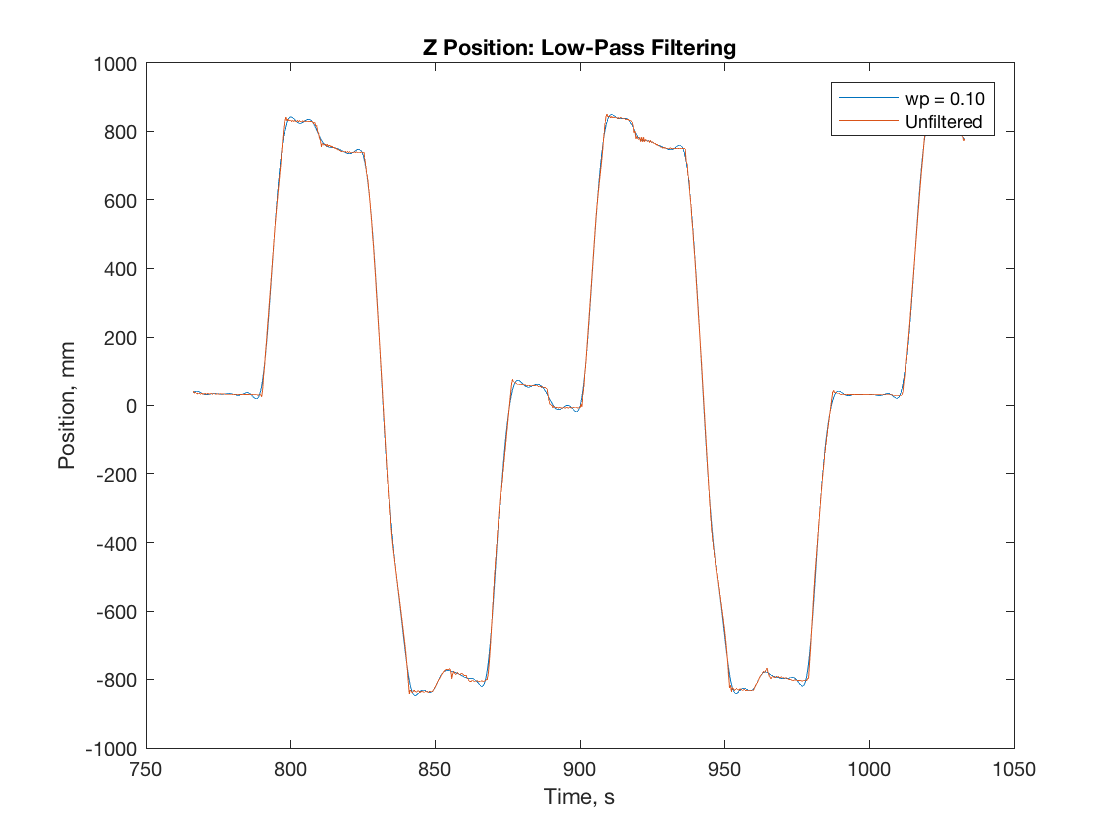


Figure 6: Low-pass filtered data, wp = 0.1 Hz.

This filter has the desired effect – the small step changes in position are clearly visible. While further tuning of this filter’s parameters will be required as we continue to design our stop-detection filtering, we now have a robust low-pass filter.

We now turn to detecting when the robot has stopped at different waypoints. Fundamentally, we are trying to detect a first derivative: we can do this effectively with a first-order Gaussian derivative filter[[1]](#footnote-1). The largest challenge that we face when detecting when all three axes have stopped moving is that for moves where a given axis moves very little – e.g., in the move across the top of the pattern, where the Z and X signals change very little – we want to ensure that we are able to detect two distinct stopped states. We can always use the third axis as a reference to detect stoppage, but a robust system will ideally be able to identify stopped states in all three axes.

We create a 1D Gaussian filter using the fspecial command, and calculate its derivative. We then convolve this with our signal, and truncate the leading and trailing terms to realign our filtered signal with our original signal. The output of this process is shown in Fig. 7 below:

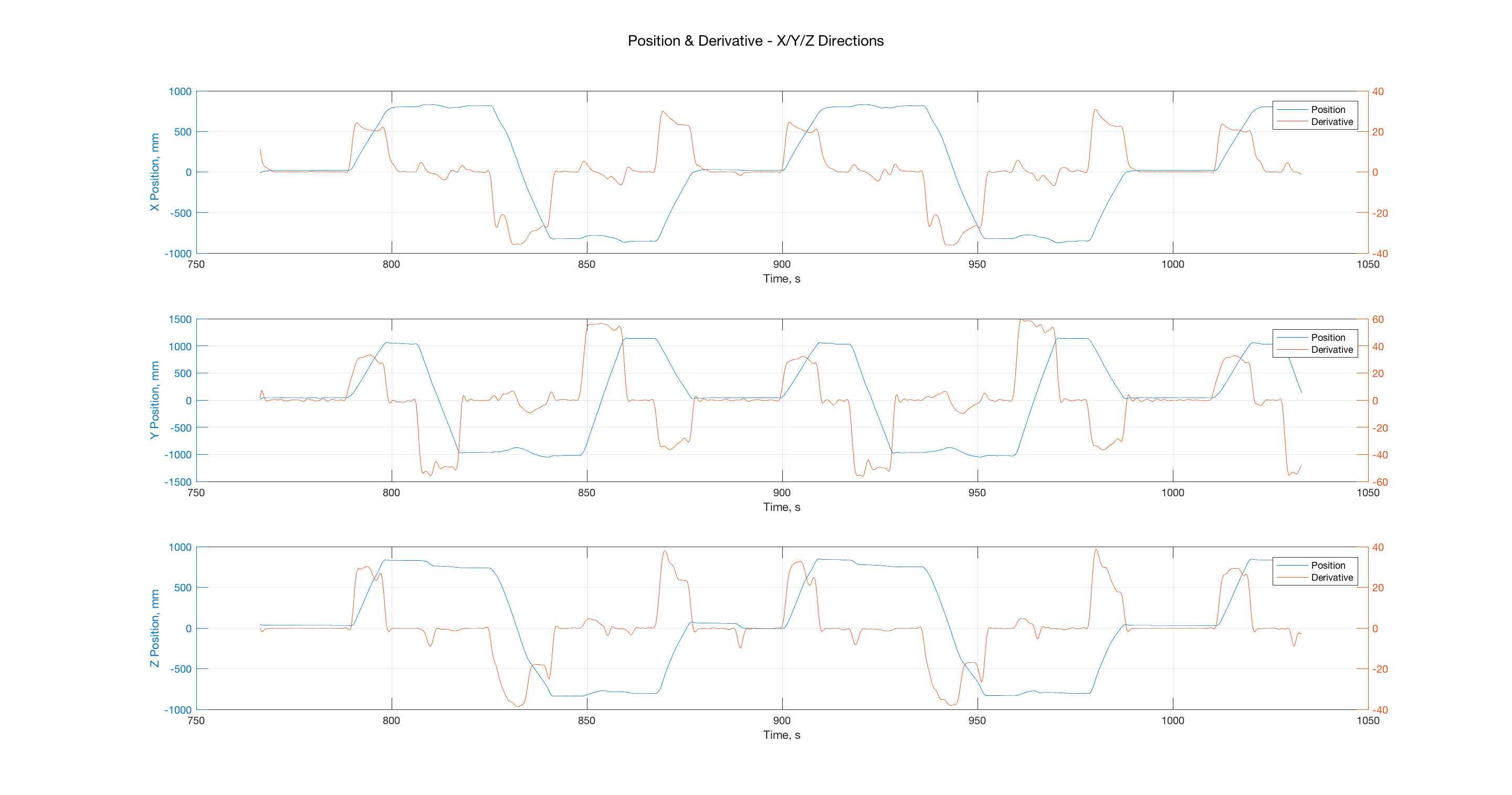


Figure 7: Plots of axis position and derivative, X/Y/Z axes

The Gaussian filter works quite well for this purpose. We can now analyze the filtered signal to determine what threshold for “stopped” behavior we should set, and use that threshold to determine when the robot has reached various waypoints.

Detecting our “stopped” state presents another interesting filtering problem. We first compare our position derivative signal generated with the Gaussian filter to some threshold, and generate a logical signal for each axis indicating whether the derivative is below the threshold (e.g., stopped) or not. However, this method also detects momentary zero crossings – for instance, when motion in a given direction reverses if the robot overshoots. These overshoots are generally quite small, on the order of one to two samples. To eliminate these, we design a new type of averaging filter. This filter works by averaging a number of samples of our logical signal around a point, and then performing a floor() operation on the resulting average to ensure that it is logical as well. The filter average window is moved along the signal using a FOR loop, in a process similar to convolution. The output of this filter is shown below in Figure 8.

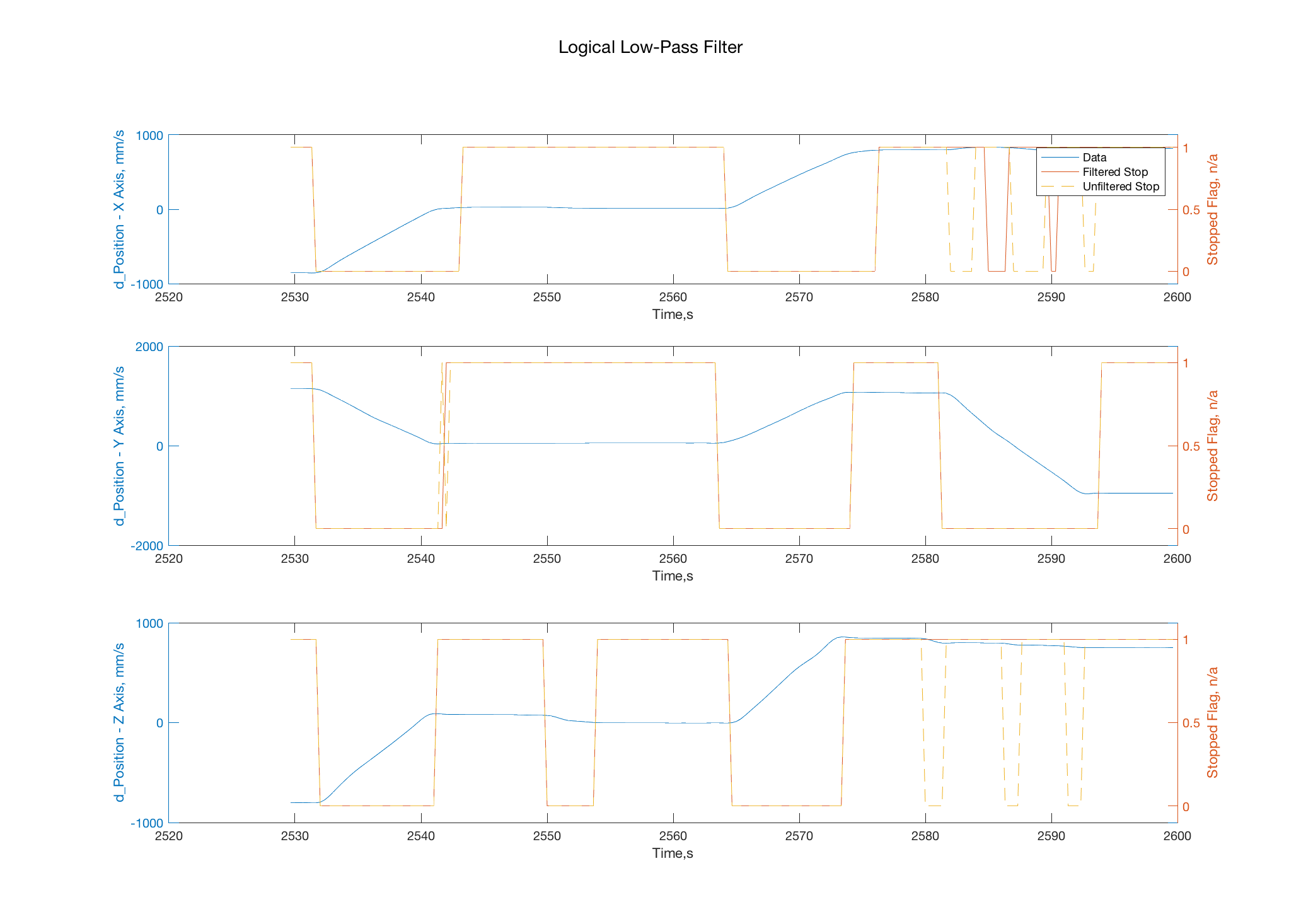


Figure 8: Output of logical low-pass filter. Filtered output in red, unfiltered input in yellow.

Interestingly, this filter appears to introduce no discernable phase lag for clearly defined signals (e.g., the filtered and unfiltered signals start simultaneously for well-defined stable states), while still performing a reasonable smoothing function.

We now need to determine when all three signals have reached a “stopped” state, to detect when the entire vehicle is stopped. Because the previous operations have produced a logical signal, we can now perform a logical comparison to determine when all three signals are stopped. We generate signals that indicate a stopped state, and also that indicate state transitions (moving-stopped and stopped-moving).

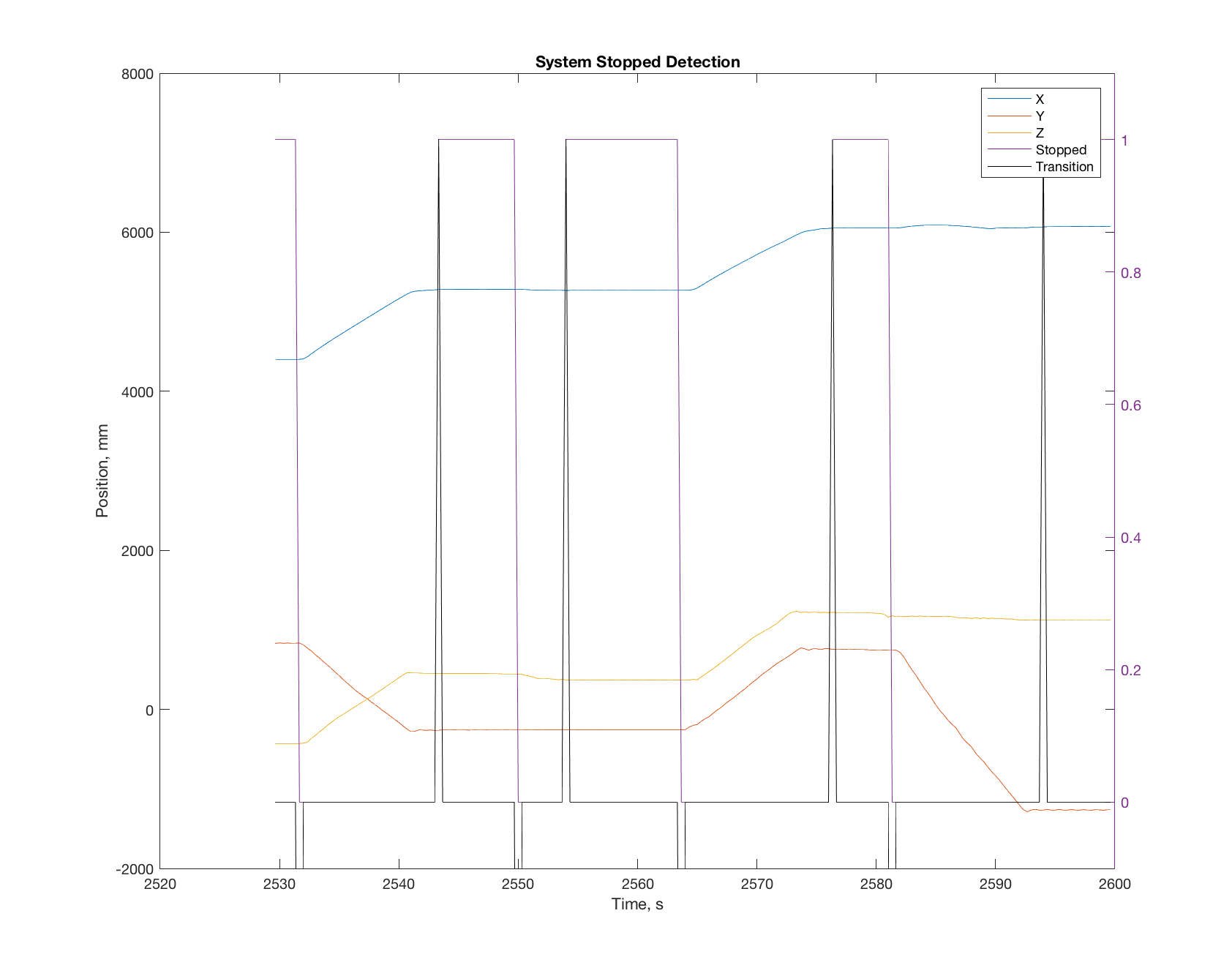


Figure 9: System Stopped Detection

With this information, we now turn to detecting how many distinct stopped states there are (e.g., waypoints), and set up indexes of which run we are on and which waypoint we are at. We first identify how many distinct stopped regions we have by iterating through the stopped regions measured above; calculating average positions at the stopped state for each axis; and then comparing that average position while stopped to all previously measured stopped-state average positions. We assume that the trajectory the system moves through is periodic, so once a repeat has been found, we stop testing new stopped states. We then create index vectors for our dataset which track which waypoint each stopped state corresponds to, and which full cycle of waypoints data belongs to.

Unfortunately, this process is not completely robust. As Figure 10 below shows, there are points in the trajectory where a stopped state is not detected, or where the stopped state is interrupted and the waypoint count is incremented. Figure 9 shows this as well, where the stopped state between t = 2545 and t = 2565 is split into two segments.

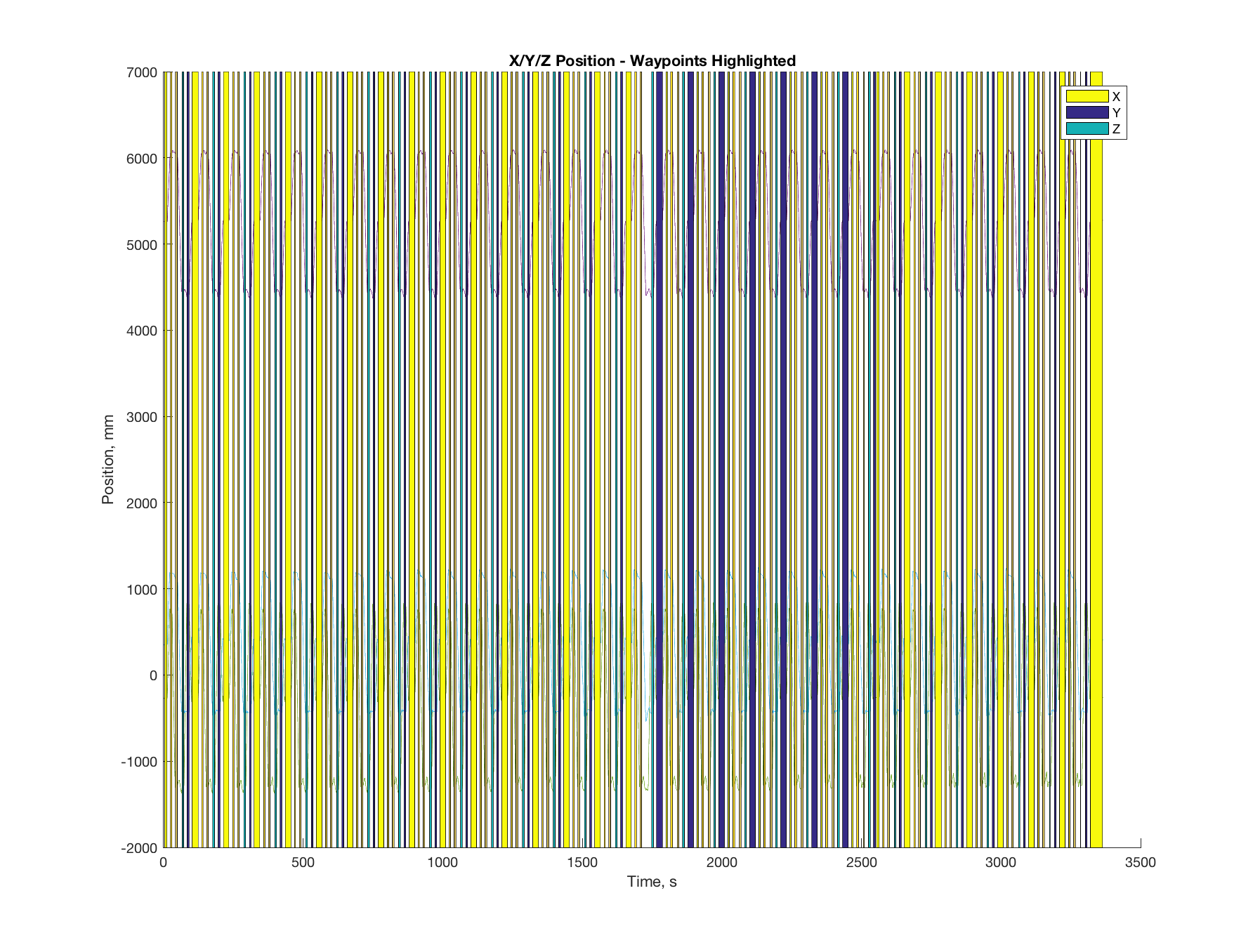


Figure 10: Initial Waypoint Highlighting

There are a number of ways to address these two issues. Waypoint double-counting, particularly, could be addressed by determining waypoint number based on comparison of a given waypoint to the set of known waypoints, instead of blindly cycling through the waypoint count. Missing waypoints could also be addressed by adjusting filter parameters. However, for this experiment – and given that the number of missed/double-counted waypoints is small – we simply add these waypoints in manually.

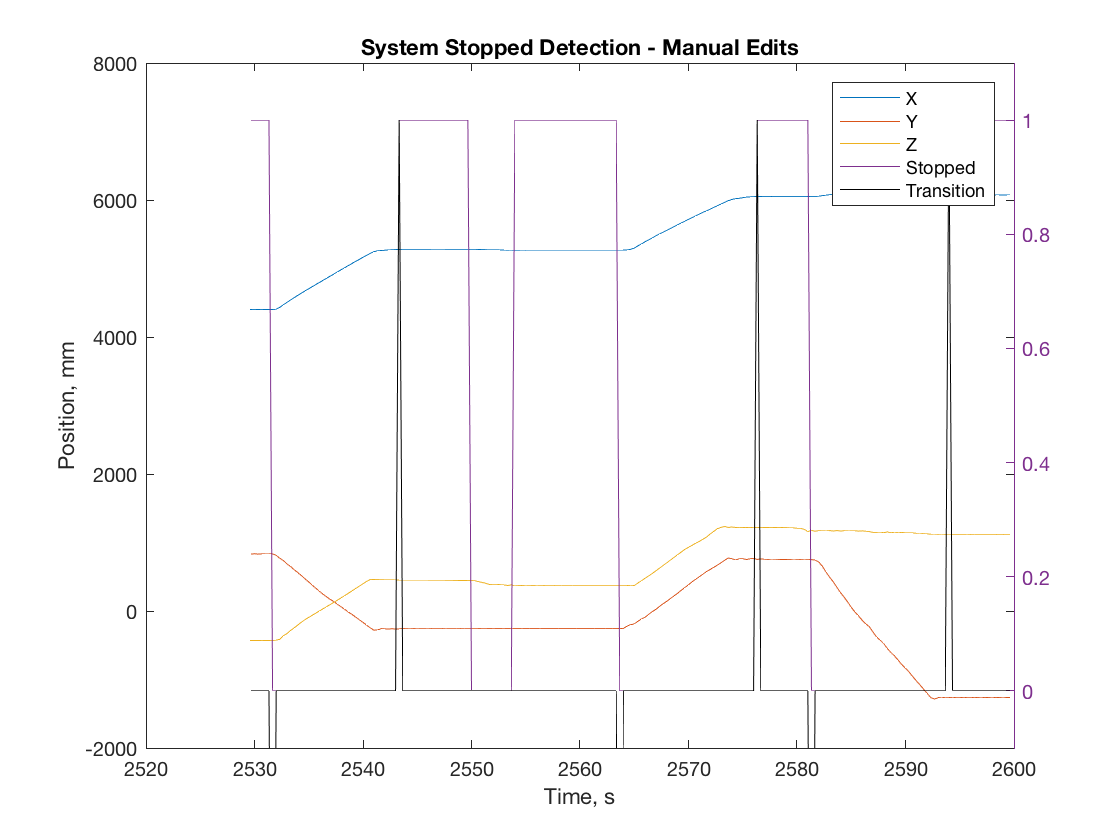


Figure 11: System Stop Detection, post edits. Note that no transitions are recorded at t = 2550 and t = 2554.

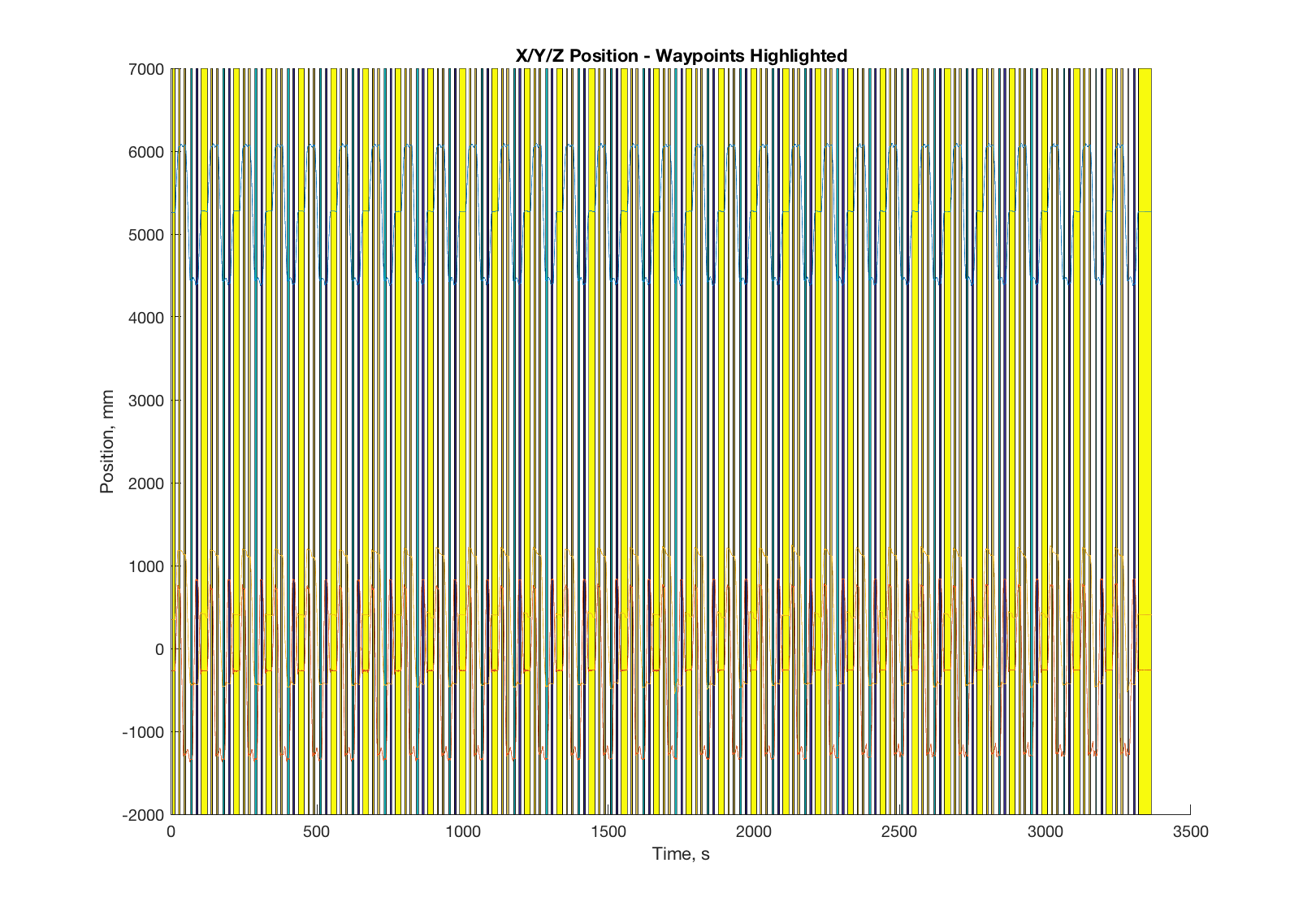


Figure 12: Final Waypoint Highlighting

We now have a dataset that is ready for processing, comprising X/Y/Z position data; time; stopped state indicators; waypoint numbers for each stopped state; and cycle count. For this dataset, we measure 5 distinct waypoints, and 30 cycles through these waypoints.

Finally, we turn to the problem of generating new datasets. We create a simple tool that allows the user to specify a series of waypoints, along with other run parameters such as time-per move, dwell time, and number of cycles. They may also inject a sinusoidal signal of controllable amplitude and frequency into the generated dataset, as well as white noise defined via SNR. Examples of generated data input and output are shown below:

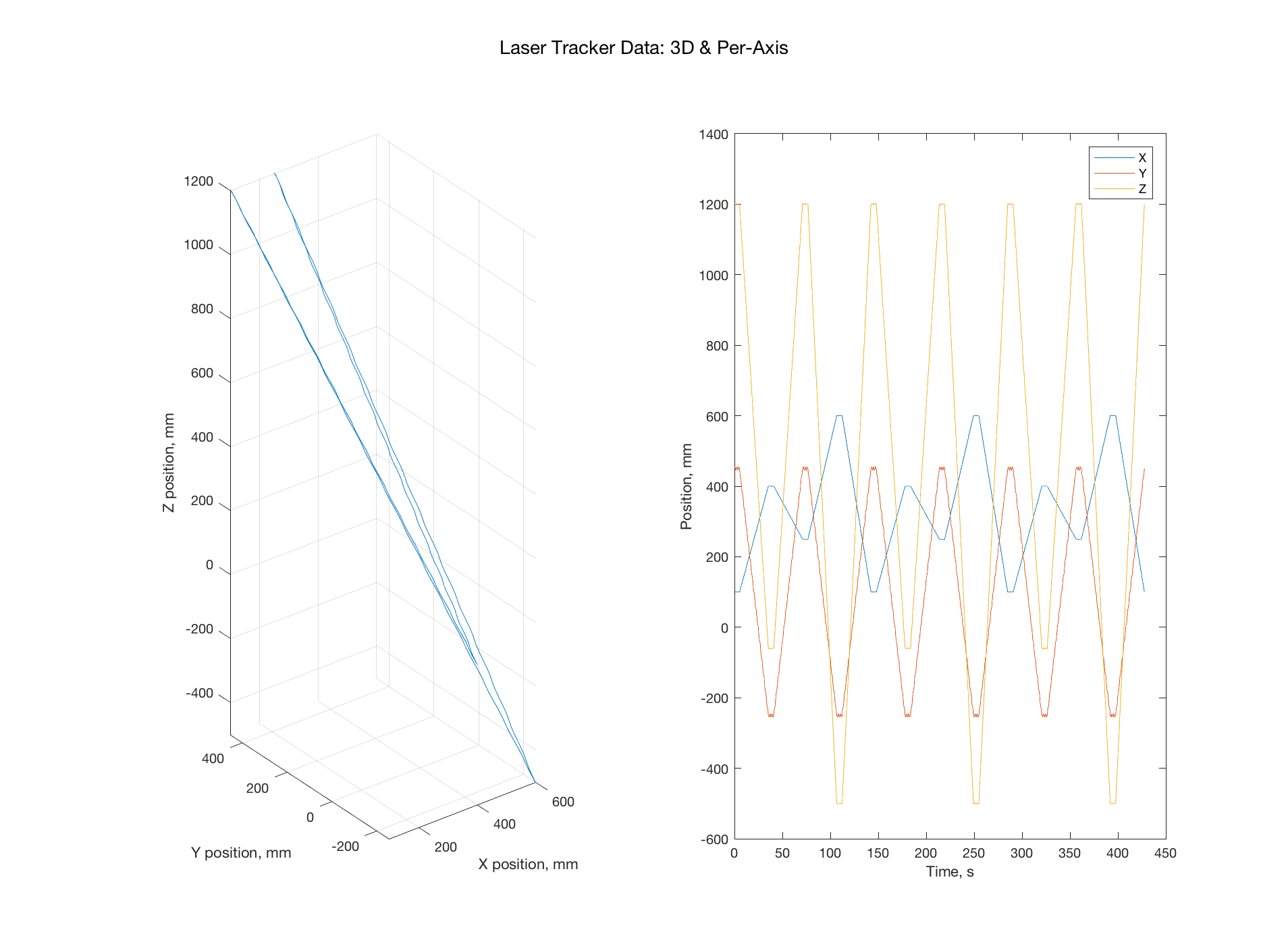


Figure 13: Generated Dataset – 3D and Per-Axis Movement

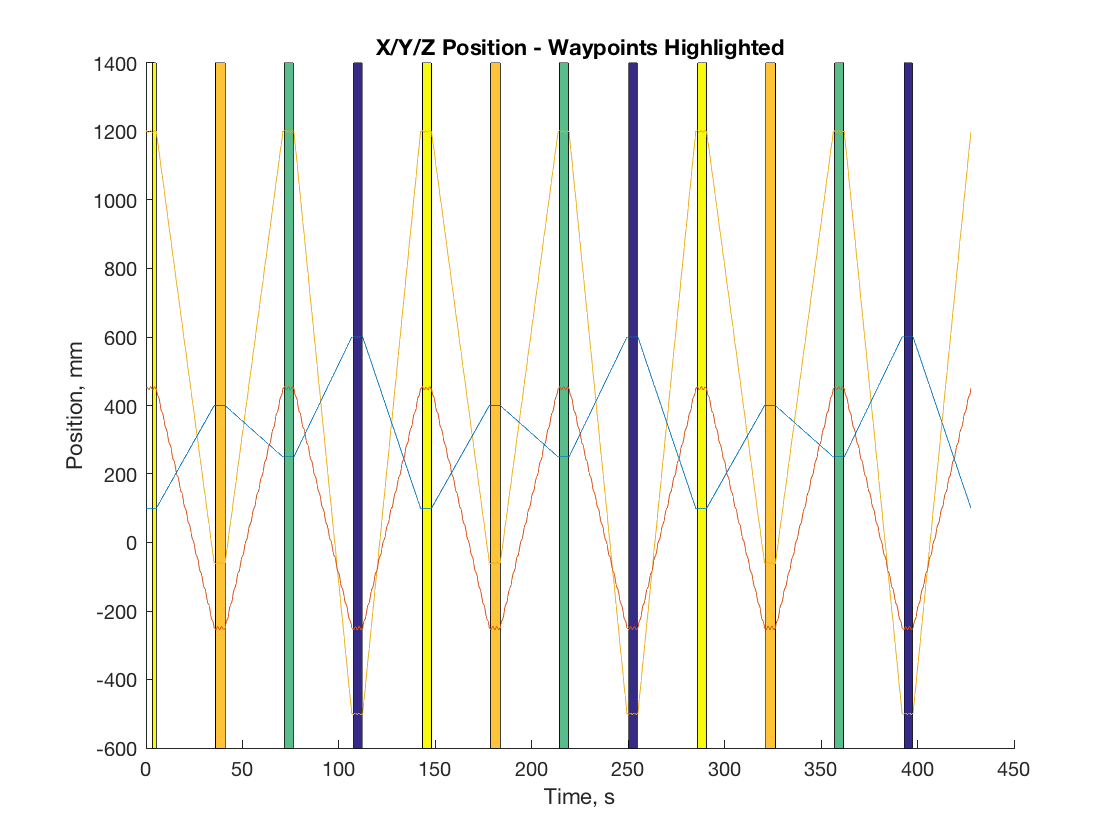


Figure 14: Waypoint Highlighting – Generated Data

As expected, some adjustment of filter parameters is required when new data is analyzed. The filtering toolchain developed here seems to be particularly sensitive to constant single-frequency noise present in the signal, with stopped states getting missed if the amplitude of the signal is too high. This is likely because of the derivative filter + simple binary threshold method that is used to determine a “stopped” state – this method performs poorly if the system is vibrating rapidly at a small amplitude. However, the value of this dataset generation tool has been clearly demonstrated, as it allows insights into the performance of the filtering toolchain and indications of how it might be improved.

1. It should be noted that the Gaussian filter applied in this manner also provides some low-pass filtering: however, we can adjust this by changing our sigma parameter. [↑](#footnote-ref-1)