Mid-Movement Error Corrections After Visual and Haptic Perturbations Reflect Latency and Performance Differences

Pablo Gonzalez Polanco
Marquette University

Follow this and additional works at: https://epublications.marquette.edu/theses_open

Part of the Engineering Commons

Recommended Citation
https://epublications.marquette.edu/theses_open/681
MID-MOVEMENT ERROR CORRECTIONS AFTER VISUAL AND HAPTIC PERTURBATIONS REFLECT LATENCY AND PERFORMANCE DIFFERENCES

by

Pablo Gonzalez Polanco

A Thesis submitted to the Faculty of the Graduate School, Marquette University, in Partial Fulfillment of the Requirements for the Degree of Master of Science

Milwaukee, Wisconsin

August 2021
ABSTRACT

MID-MOVEMENT ERROR CORRECTIONS AFTER VISUAL AND HAPTIC PERTURBATIONS REFLECT LATENCY AND PERFORMANCE DIFFERENCES

Pablo Gonzalez Polanco, B.S.

Marquette University, 2021

We examined a key attribute of sensory motor skill: the capability to adjust ongoing actions to correct performance errors that arise during the execution of a task. Fifteen subjects grasped the handle of a horizontal 2D planar robot that enacted a small viscous resistance to the arm movement. The subjects initiated interception movements using a prompted Go cue to quickly catch a pseudo-randomly moving target. On some trials, the robot viscosity field or the target’s motion (speed) was altered without any warning at the time of the Go cue. The viscosity could be increased which made it more difficult to move the handle or decreased which made it easier to move. Target speed could also be increased or decreased. We analyzed arm movement kinematics fitted to a sum-of-Gaussians model to determine if an error correction occurred within the interception attempt, and to quantify its timing and magnitude. We found that corrections during increasing viscosity perturbations occurred sooner (154ms) than on control trials (215ms), while perturbations during decreasing viscosity perturbations had longer latencies (272ms). By contrast, we found that visually perturbed trials had similar error correction latencies to control trials. We also found that the magnitude of the initial error correction adapted to the environmental conditions in each trial, with speed/viscosity increases eliciting more vigorous responses and speed/viscosity decreases eliciting less vigorous responses than control conditions. Finally, we found that corrections, whether they were generated internally or due to haptic or visual perturbations, were performed early in the reaching movement before sensory feedback could indicate that the target had been captured or missed. These results are consistent with models of motor control where error corrections are done in response to predicted performance rather than actual performance.
ACKNOWLEDGMENTS

Pablo Gonzalez Polanco

In no particular order, I would like to thank my mother, my father, my sister and my girlfriend Kelly McDonnell for their support. I would like to thank my teachers, my committee and my advisors Dr. Scheidt and Dr. Mrotek for their guidance. I would like to thank the Graduate School and all of the Marquette University administration.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS..........................................................................................i

CHAPTER

1. RATIONALE AND SPECIFIC AIM........................................................................3
2. BACKGROUND.....................................................................................................5
3. EXPERIMENTAL STUDY....................................................................................14
   3.1. Introduction..................................................................................................14
   3.2. Methods.......................................................................................................17
   3.3. Results........................................................................................................29
   3.4. Discussion...................................................................................................38
4. LIMITATIONS AND FUTURE DIRECTIONS..................................................47
5. BIBLIOGRAPHY.................................................................................................49
6. APPENDICES.....................................................................................................53
   6.1. Appendix A: Preparatory Cue and Go Cue Arduino Code ............53
   6.2. Appendix B: Robot Arm and Screen Display Code .................56
   6.3. Appendix C: Sum of Gaussians .........................................................86
CHAPTER 1: Rationale and Specific Aim

This project advances the understanding of a key attribute of sensorimotor skill: the ability to adjust ongoing actions to correct for performance errors that arise during the execution of a task. Take for instance a tennis player returning a serve: the player must use visual information and make cognitive predictions to guide the racquet and perform a successful return. During the swing the tennis ball could unexpectedly change direction due to a gust of wind, and the player would have to quickly predict the new ball trajectory and skillfully correct for it in order successfully return the ball. Or the player could inadvertently bring a heavier racquet to the game that day and need to modify the applied force (or "vigor" of movement) to accomplish each swing correctly.

Our goal is to understand how the brain adjusts the timing and vigor of ongoing actions to compensate for three distinct sources of performance error. Some errors, like the tennis ball example, are caused by unexpected target motions sensed visually. Others, like the racket example, are caused by unexpected changes in hand-held loads sensed haptically. Yet other arise spontaneously due to errors in the planning and execution of movement. Are error corrections in response to unexpected visual perturbations or unexpected haptic changes in the environment different from the changes required from an internally generated error? Do error corrections in each case reflect compensations for actual realized
performance errors or might they reflect anticipatory responses to predicted errors? Thus, my specific Aim is:

**Specific Aim:** To characterize the timing and vigor of mid-movement error corrections that arise during a time-constrained manual interception task in response to unexpected target speed changes, in response to unexpected changes in a hand-held mechanical load, and in response to target capture errors that arise spontaneously due to errors in the planning and execution of the initial interception attempt.

Based on recent literature, we expect haptic perturbations to the arm may trigger faster mid-movement corrective responses than those generated internally or by visual perturbations. We also expect error corrections may be driven by predictions of performance error rather than of actual performance error, with corrections beginning early in the hand motion, i.e., before the hand would reach the target or sensory confirmation of a successful target capture could be received. Better understanding of error correction could lead to improved training approaches wherever performance optimization is desired, whether in athletics or recovery from neuromotor injury. Using knowledge about how error corrections typically occur in healthy adults, training regimes could be developed to perturb situations with specified magnitudes and specified times to improve the executive functioning associated with improving performance.
CHAPTER 2: Background

Because mistakes are made during the production of skills, mid-movement error corrections are critical for nearly every goal-directed, physical interaction with a dynamic and unpredictable world. Often these errors must be corrected immediately, otherwise the skill will fail. For example, one may need to adjust the hand trajectory while attempting to swat a fly that moves unpredictably so that the fly can be caught. Errors can also occur due to unexpected environmental factors such as wind, uneven or slippery surfaces, lighting changes, etc. These corrections require fast and precise action from our sensorimotor system and are essential for every goal directed task; whether it is hitting a tennis ball, avoiding falling on slippery ice, or compensating for motor deficits after neuromotor injuries such as stroke (Dipietro, Krebs, Fasoli, Volpe, & Hogan, 2009; Fisher, Winstein, & Velicki, 2000; Krebs, Aisen, Volpe, & Hogan, 1999; Scheidt & Stoeckmann, 2007). But how do we study them?

Real world tasks are not ideal for studying error correction since it is very difficult to control the timing and magnitude of performance errors in uncontrolled environments. Highly controlled laboratory tasks, where subjects have little room for decision making or to make mistakes, also do not work very well. Examples of these tasks are pointing at targets when they appear on a screen. Here error corrections (if there was any) would be small and difficult to detect. In contrast, a laboratory task that involves
strategy and decision-making such as manual interception (Georgopoulos, Kalaska, & Massey, 1981; Soechting & Lacquaniti, 1983) can be used effectively to control timing and magnitude of events that produce performance errors, thus facilitating their study. Manual interception underlies many activities that enhance quality of life, from hitting a tennis ball to reaching for a grandchild’s hand.

*Manual interception, and on-line corrections for performance errors.*

Manual interception refers to movements of the hand or a hand-held tool in order to catch, hit, or push a moving target. Early studies of manual interception involving rhesus monkey were performed by Georgopoulos, Kalaska, & Massey (1981). In their study the monkeys were trained to use an articulated manipulandum to move and capture static targets on a planar surface. In some trials, the target location could change unexpectedly 50 to 400 ms after the target was presented. During these conditions Georgopoulos et al. found that monkeys were able to correct their hand trajectory mid-movement and reach towards the new target location shortly after the perturbation was presented, giving us some of the first inklings into the workings of on-line error corrections.

Many follow up studies have been done to further the understanding of mid-movement error corrections. Mrotek 2013 sought to understand how the hand and eye control systems interact during target speed perturbations. In her study, subjects were instructed to use their finger to intercept targets moving pseudo-randomly on a 2D plane after a
Go signal (Fig 1). On 2/3 of trials, the target speed increased or decreased at various times after the Go cue. When the speed perturbation occurred close to the Go cue time, subjects were able to incorporate the new target speed into their motor plan. However, for perturbations occurring later (once the finger had already started moving) subjects began to error correct 150-200 ms after the speed perturbation.

Figure 1: adapted from Mrotek 2013. Figure depicts hand movement during a manual interception error correction. Dotted lines: Unseen target paths; Gray arrows: direction of target motion. Filled red circle: target location at the GO cue; Red ring: cursor location 150 ms after the GO cue. Green dashed line: initial hand movement direction. Green solid line: realized hand movement direction. Blue arrow: point at which the direction change happens in response to the speed perturbation.
Generally, interception movements are directed towards the future location the target will occupy (Mrotek, 2013; Mrotek & Soechting, 2007). If the target is moving in a predictable manner subjects tend to aim far ahead into the predicted target location, while if the target displays an unpredictable movement pattern subjects will aim closer to the where the target is but move faster to intercept it (Daum et al. 2007). Anticipation implicates use of explicit and/or implicit predictions of the object’s motion (for a review see Brenner & Smeets, 2018).

**Sources of performance errors and their latencies**

In studies of multisensory control of motion (such as manual interception) subjects who make an error are typically under severe time-constraints to correct it; the difference between success and failure depends on how quickly the performance error is sensed and transformed into appropriate adjustments to the ongoing motor behavior. There is urgency on the part of the subject, thus this type of task can test the limits of sensorimotor information processing speed related to movement planning and execution in the presence of several different forms of task uncertainty.

In some cases such as horizontal planar reaching, sensorimotor performance can be dominated by vision (Flanagan and Rao, 1995) even in the presence of visual uncertainty (Judkins and Scheidt, 2014). Studies using manual interception to investigate error correction have shown that the visual characteristics of the target (such as texture, luminance, color,
orientation etc.) can affect how quickly we recognize and correct movement (Veerman et al. 2008; Brenner & Smeets 2009). In other cases, such as limb stabilization, somatosensory cues can have the greater impact in the success or failure of the task despite proprioceptive uncertainty (Lederman et al., 1986). Yet other studies report motor task performances reflecting a cooperative weighting of sensory cues, similar to perception (Cluff et al. 2015; van Beers et al. 2002), arguing that we use the most “up-to-date” sensory information to perform error corrections and judge their urgency (Wijdenes et al. 2011). As the literature in this area is unequivocal as to the contributions of visual and somatosensory information to the generation of mid-movement error corrections, it is important that we evaluate the differences in the developed responses when visual and somatosensory cues are perturbed.

In a recent study conducted by Camponogara et al. (2019) participants were instructed to move their right hand a distance of 50 cm so as to grasp an object. The experiment was divided into three somatosensory conditions: In the visual condition, subjects could see the object while performing the grasping motion with their right hand. In the haptic condition subjects were not able to see the object but could touch it with their left (non-grasping). In the visuo-haptic condition subjects could both see the object and touch the object with their left hand. On some of the trials, the size of the held object could be perturbed in the middle of the reaching movement and the latency of the corrective response
prompted by these perturbations was measured using kinematic data of the right hand. The results indicated that perturbations during the haptic conditions had shorter correction latencies (130ms) than visual conditions (171ms) and similar to visuo-haptic conditions (144ms). Findings using EMG data instead of kinematic data to measure error correction latency Cluff et al. (2015) report similar findings across multiple studies, with visual error correction latencies emerging ~100ms after perturbations, and error correction to mechanical perturbations happening ~60 ms.

The use of kinematics for submovement modeling of error corrections

A key question in the study of error corrections is how to detect and measure them. Some have used hand movement displacement during reaching tasks as a way to detect initiation of the corrective movement (Mrotek 2013, Brenner and Smeets 2009, Veerman et al. 2008), while others have used EMG signaling to detect changes in muscle activity (Debicki and Gribble 2004, and Kutzer, Pruszynski and Scott 2009). A third way of describing hand movements is the use of overlapping submovements to break down their kinematic profile. In 2000, 2002 and 2003 Novak, Miller, and Houk performed a series of experiments were subjects had to rapidly rotate a knob to align its pointer with several targets. Through this task they were able to break down the hand movement into three different movement types: a primary (initial) movement, and two types of submovements: overlapping submovements, which were initiated before the primary movement ended; and
nonoverlapping submovements, which happened after the primary movement ended. Since the overlapping submovements could be small and difficult to detect they used a “test of movement regularity” to examine the number of jerk and snap zero crossings during the second half of the movement to determine the number of hand submovements in any given trial. Novak et al. reported that these submovements were done as compensatory corrections when subjects failed to estimate the proper amplitude and vigor in their initial hand movement. A shortcoming of using hand speed to detect submovements during manual interception tasks is that applying forces to objects that have non-negligible inertia - such as a robot handle - tend to cause motions that out-live the forces that produced them. Thus, the number and magnitude of submovements derived from snap and jerk zero crossings to hand speed profiles can be artificially inflated.

The role of prediction

To move accurately, the brain must transform sensory feedback into motor commands that initiate movement at a proper time and in a way that is robust against errors caused by motor variability and external disturbances (Cluff et al. 2015). Initially, when subjects take on a challenging motor task, their hand trajectories will be distorted and they will tend to make mistakes in their initial movements (Kawato 1999, Flanagan et al. 2003, Novak et al. 2003). These errors are referred to as
internally generated or self-generated errors and arise from mistakes in the initial motor plan.

Take for instance a manual interception task using a robotic manipulator that generates a constant viscous field. Internally generated errors would include failing to estimate the future target location, the appropriate amount of force to move the arm, the correct time to move, or the correct direction in which to move. As mentioned earlier, other types of errors include externally generated errors such as visual error (i.e. those due to unexpected changes in target speed) and haptic errors (i.e. those due to unexpected changes in the force field rendered by the robotic manipulator). Several research groups (e.g., Desmurget and Grafton, 2000, Kawato 1999, Wolpert and Miall 1996) have argued that our central nervous system (CNS) internally predicts the sensory outcomes of its motor commands and compares those predictions against actual sensory feedback to correct for errors in the ongoing motor plan. By doing so, the CNS would not need to wait until it had received positive sensory confirmation that its intended action had failed. Rather, error corrections could begin in anticipation of an impending failure, potentially leading to greater likelihood of initial success, and faster achievement of success when error corrections are required.

My thesis project uses manual interception to explore the magnitude and timing of error corrections that arise due to three different types of performance errors: those resulting from internally generated
errors of movement timing, direction, and/or vigor; those due to unexpected changes in target speed; and those due to unexpected changes to the mechanical characteristics of the hand-held load. I chose to look at these different kinds of error corrections in the context of manual interception to gain insight into how the brain processes sensorimotor information to guide the online control of goal-directed actions. I expect that for internally generated errors, subjects would need to predict a new target location and to use that prediction to plan and execute an error correction. For visually generated errors, the subject would additionally need to estimate the new target speed and use it to plan the error correction. Finally, for haptic perturbations, the subject would additionally have to estimate the new limb/environmental dynamics and use that "internal model" to adapt the vigor of their motor plan so that the hand may arrive at the newly updated goal location at the right time. I expect that the different sources of performance error will give rise to systematic differences in the timing of error corrections during manual interception, reflecting differences in the type of information processing required to plan and execute error corrections mid-movement.
CHAPTER 3: Experimental Study

3.1 Introduction

One of the key factors that allows us to perform accurate goal-directed movements is our capability to adjust ongoing actions to correct for performance errors that arise during the execution of a task. We live in an unpredictable world, and as such, a prearranged motor plan can become compromised in the face of an unexpected event. A person trying not to slip on ice, a bright light blinding a goalkeeper's vision while trying to stop a goal, or a fly moving unpredictably while trying to catch it are all cases where, to be successful, one must quickly and skillfully adapt to the unexpected environmental perturbations by correcting for any errors in the motor plan. But how quickly can we correct for these perturbations? And moreover, do responses to errors generated by visual or haptic perturbations differ from responses to errors generated internally?

Manual interception studies (Georgopoulos, Kalaska, & Massey, 1981; Soechting & Lacquaniti, 1983) have been used effectively to control timing and magnitude of events that produce performance errors, facilitating their study. The importance of visual perturbations during these studies has been widely examined (for a review see Brenner & Smeets, 2018). These perturbations introduce changes to the target, be it changes in speed, position, size, shape, luminance, or orientation in order to trigger corrective responses in the subject. Across these studies, latency of the
error correction varies depending on the type of task, the type of visual perturbation and the timing of the perturbation. However, when examining arm force or motion changes in response to these perturbations, error correction latencies typically occur between 150-200ms (Brenner and Smeets 2008; Mrotek 2013; Veerman, Day and Lyon 1999; Cluff et al. 2015).

The effect of haptic perturbations during error correction studies has also been of great interest to the field of neuromotor control (for a review see Pruszynski and Scott 2012). From studies where arm kinematic data is used to measure error correction latency it is common to see differences of 120-180 ms from when the perturbation is introduced to the correction (Cluff et al. 2015). In the Camponogara et al. (2019) study, participants were instructed to reach and grasp an object with their right hand. In some cases the object would change sizes in the middle of the grasping movement and the subjects would be able to detect the perturbation either visually, haptically (left-hand) or with both senses at the same time. They found that haptic detection of perturbations elicited faster error correction responses than visual perturbation and similar to visuo-haptic ones. Even though it seems that corrections from haptic perturbations have faster latency responses than from visual perturbations, few manual interception studies have investigated both types of perturbations along with internally generated errors (errors arising
from flaws in our initial motor plan) in the same task and using the same methodology for timing the onset of the error correction.

To determine whether corrections during visual perturbation and haptic perturbations differ from corrections triggered by self-generated errors, we asked participants to perform a manual interception task toward a pseudo-randomly moving target using a 2D planar robot after a predictable Go cue. On some trials, at the same time as the Go cue, we introduced target motion speed changes or abruptly altered the environmental viscosity of the 2D planar robot. We sought to test two hypotheses. First, we tested the extent to which arm movement error corrections during manual interception reflect compensations for predicted performance errors rather than responses to actual performance errors (FishBach et al. 2007). Forward models of motor control argue the existence of internal neural mechanisms that can allow us to estimate the future target and hand trajectories, allowing us to make corrections in our movements based on the expected outcome (Desmurget and Grafton 2000; Kawato 1999, Wolpert and Miall 1996). If these models are correct, we expect that error corrections in our experiment will be detected early in the hand trajectory and the subject will rectify the movement before the target would be reached if the initial hand movement had been successful or sensory confirmation of a successful target capture was received. Secondly, we tested the hypothesis that the magnitude and timing of mid-movement error corrections reflect differences in the neural information
processing required to infer the source of pending performance errors (whether they are self-, target-, or environment-generated) and to compensate appropriately for those errors. If findings from Camponogara and Volcic (2019) grasping adjustments transfer to manual interception, we expect haptic perturbations to the environment to trigger a faster mid-movement corrective response than those generated internally or by visual perturbations.

3.2 Methods

Participants

Fifteen healthy subjects [age 20-30 years; 5 males and 10 females] participated in this study. All subjects had normal or corrected-to-normal vision, and none had known neurological deficits. All subjects provided written informed consent to participate in experimental procedures that received institutional review and approval in accordance with the Declaration of Helsinki.

Experimental Setup

Each subject sat in a high-backed chair and grasped the handle of a horizontal planar robot with their right (preferred) hand (Fig 1A). The robot was actuated by two brushless DC torque motors (M-605-A Goldline; Kollmorgen, Inc. Northampton, MA). A 16-bit data acquisition board (PCI-6031E DAQ; National Instruments Inc., Austin, TX) sampled analog force from a load cell (85M35A-I40-A-200 N12; JR3 Inc.,
Woodland, CA) mounted under the handle. Handle location was resolved within 0.038 mm using joint angular position data from two 17-bit, encoders (A25SB17P180C06E1CN; Gurley Instruments, Troy, NY).

The robot motors rendered nominal hand forces that made the robot's handle feel like it was moving through a lightly viscous fluid (Eqn. 1):

\[
\begin{bmatrix}
F_x \\
F_y
\end{bmatrix} = \begin{bmatrix}
-\mu & 0 \\
0 & -\mu
\end{bmatrix} \begin{bmatrix}
V_x \\
V_y
\end{bmatrix}
\]

[1]

where \(F_x\) and \(F_y\) are forces felt at the hand, \(\mu\) is a constant scaling parameter (40 Ns/m), whereas \(V_x\) and \(V_y\) are components of the hand velocity vector. Motion under this isotropic viscous field induces stabilizing hand forces applied in a direction directly opposing movement. Robot control and data collection were performed at 1000 samples per second. Handle kinematic data and robot control signals were stored to disk for post-processing.
Figure 1: A) Experimental set up: subjects sat in the high back chair and made reaching movements while holding the handle of a horizontal planar robot. A projector was positioned above the horizontal display to show visual targets on top of the horizontal screen. B) The 5 paths created for the target movement. Target motion could start from one of two locations along each path yielding 10 unique target trajectories. GO cue timing was selected to yield initial target capture movements into 10 different directions spanning the full 360° range about the hand's starting position.

A horizontal display screen was mounted 2 cm above the robot handle, occluding the vision of the arm and hand. Ongoing visual feedback of the hand motion was provided at all times via a circular cursor (2 cm diameter) projected on the horizontal screen. A starting “home” target was also projected on the display screen in the middle of the subject's reachable workspace; this home target was always visible on-screen throughout the experiment.

During each trial of the experiment, a 2 cm diameter circular target moved smoothly around the screen along a seemingly random path designed to take 10 s to complete. The target motion was determined through a sum-of-cosines method (Eqn. 2):

\[
\begin{bmatrix}
X(t) \\
Y(t)
\end{bmatrix} = \begin{bmatrix}
a_1 & 0 & a_2 & 0 \\
0 & b_1 & 0 & b_2
\end{bmatrix} \begin{bmatrix}
\cos(\omega(t + t_{offset})) \\
\cos(\omega(t + t_{offset})) \\
\cos(h_x \omega(t + t_{offset}) + \phi_x) \\
\cos(h_y \omega(t + t_{offset}) + \phi_y)
\end{bmatrix}
\]  

[2]
where $X(t)$ and $Y(t)$ are the horizontal and vertical positions of the target at time $t$, whereas $h_x$ and $h_y$ represent spatial frequency harmonics (ranging from 2 to 5) and $\phi_x$ and $\phi_y$ represent phase variables that determine the overall shape of the path. Parameter $t_{offset}$ allows for variable starting points along each path. Using Equation 2, cursor speed changed in a natural way as the target moved along its path (i.e., speed was faster along straighter path segments and slower along the curved segments). In each trial, the target motion followed one of 5 paths and each path target motion could begin at 2 different starting positions, creating 10 unique movement patterns (Fig 1B; Table 1).

<table>
<thead>
<tr>
<th>Path</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>a₂</td>
</tr>
<tr>
<td>1</td>
<td>2.70</td>
</tr>
<tr>
<td>2</td>
<td>2.70</td>
</tr>
<tr>
<td>3</td>
<td>2.70</td>
</tr>
<tr>
<td>4</td>
<td>3.00</td>
</tr>
<tr>
<td>5</td>
<td>5.40</td>
</tr>
<tr>
<td>6</td>
<td>5.40</td>
</tr>
<tr>
<td>7</td>
<td>3.00</td>
</tr>
<tr>
<td>8</td>
<td>2.70</td>
</tr>
<tr>
<td>9</td>
<td>4.32</td>
</tr>
<tr>
<td>10</td>
<td>4.32</td>
</tr>
</tbody>
</table>
Procedures

Subjects performed 2 blocks of 150 trials each and were required to take a 2-to-3-minute break between blocks. Prior to each trial, subjects were to move the robot handle to the home position (Fig 2A), where they were required to remain still for 1 s (Fig 2B). When the trial started, a yellow circular target appeared on the horizontal screen and began moving along one of the 10 unique paths (Fig 2C). The subject was instructed to wait at the home position and observe the target's motion until a predictable sequence of auditory and visual signals cued them to capture the target with a single ballistic arm movement (Fig 2D). If the subject missed the target, (s)he was to continue to try to catch the target until successful or until 5s had passed. Upon target capture, target motion ceased and the target's size increased by a factor of 3 to indicate success and to provide a salient reward signal (Fig 2E). After the trial ended the subject moved back to the home target and could begin the next trial.

Figure 2: Trial description: A) subject moves towards the home position; B) Once at the home position the subject waits for the trial to begin; C) Trial begins, a moving target appears on the screen. Subject observes the target trajectory; D) Go cue (Color change + auditory signal) is the indication for the subject to attempt to capture target. E) Once the subject captures the target the target expands to indicate the end of the trial.
The audible cue sequence consisted of 3 low-pitched 100 ms beeps (the preparatory cues) followed by a high-pitched 100 ms beep (the GO cue) (Fig 3A). The inter-beep interval was a predictable 333 ms, allowing subjects to plan the onset of movement to coincide with the final beep in the series. The final beep coincided with a target color change (from yellow to blue), reinforcing the instruction that subjects should attempt to capture the target using a single ballistic arm movement at that time. The purpose of providing these preparatory cues was to minimize uncertainty due to reaction times, thereby increasing experimental control over response latencies. This sequence of auditory tones could start at 0.5, 0.8, 1.1, 1.4 or 1.7 seconds after the trial started, depending on the desired initial movement direction. We selected these specific GO Cue times to direct initial target capture movements into 10 different directions spanning the full 360° range about the hand’s starting position.
On some trials, the target's speed or the robot's mechanical resistance to motion could change coincidently with the GO cue. On perturbed trials, target speed could increase by 50% (x1.5) or decrease by 34% (x0.66). Alternately, the effective robot handle viscosity could increase by 50% or decrease by 34%. Note that only 1 kind of perturbation happened on any given trial. On unperturbed trials, the target speed and robotic force field remained consistent before and after the GO cue. Thus, five trial types were possible, and subjects experienced each with 20% probability.

Subjects were initially provided 10 unperturbed practice trials to become familiar with the experimental task and the nominal robotic load.
After those trials were completed, further trials randomly alternated between unperturbed and perturbed trials. Each of the 10 unique path x 5 trial-type combinations was repeated 6 times each for a total of 300 test trials, which we subjected to detailed analysis.

Data Analysis

Hand position, hand speed and force data were low pass filtered off-line using a zero-lag, 2nd order Butterworth filter with a 20 Hz cutoff frequency (Matlab function filtfilt). We then computed hand velocity using a first-order difference equation and the mechanical power transfer at the robot handle using the vector dot product of Equation 3:

\[ P(t) = F(t) \cdot v(t) \]  

where \( P(t) \) is defined as the mechanical power transferred from the hand to the robot handle, \( F(t) \) is the vector of forces applied by the hand to the handle, and \( v(t) \) is the handle velocity vector, all as a function of time \( t \).

The hand power time series from each trial was visually inspected prior to further processing to determine whether the subject had followed instructions to capture the target with a single ballistic arm movement at the time of the Go Cue. Fourteen of the subjects complied with task instructions and attempted to capture the moving targets using ballistic reaches. The remaining non-compliant subject chose an unusual strategy
whereby she tracked the target at very low constant speed until she captured it. Data from this subject was excluded from the primary analyses described below due to failure to follow task instructions.

We used the mechanical power that subjects imparted onto the robot handle to infer the timing and magnitude of discrete target capture and error correction actions as they developed dynamically throughout each target interception trial. To do so, we fit one or more elemental Gaussian pulse profiles to the hand power time series in each trial. Our approach extends a similar approach described by Novak and Houk (2000, 2002, 2003), which fit Gaussian pulses to hand speed data to infer the presence of multiple submovements during goal-directed movements. However, using hand speed rather than hand power data to infer the presence and number of submovements is subject to an undesirable confound in that forces applied to objects having non-negligible inertia - such as a robot handle - tend to cause motions that out-live the forces that produced them. Thus, the number and magnitude of elemental actions derived from Gaussian fits to hand speed profiles can be artificially inflated. To overcome this limitation, we programmed the robot to impose a nominal (small) viscous load upon each movement. This isotropic viscous load caused the hand to come to rest when the subject ceased actively producing force against the handle. Then, by computing mechanical power transfer at the handle, we derive a signal that is positive only when the subject is actively accelerating the handle in the direction of
movement, and negative when the subject is actively decelerating the handle.

An algorithm was designed to infer the number of discrete control actions performed by each subject during each trial, as well as the timing and magnitude of those actions (see Appendix A). We refer to this approach as Gaussian effort modeling. Briefly, the number of elemental Gaussian power pulses was determined by the number of zero crossings identified in the second derivative of the hand power time series data. Generally, two zero-crossings are expected per Gaussian power pulse, although the final pulse can be truncated (and contribute only 1 zero-crossing) if the target is captured midway through that final action. The amplitude ($A_i$), center ($\mu_i$), and width ($\sigma_i$) parameters of the $i$ elemental Gaussians were obtained by minimizing the squared error between the actual power plot time series data and the sum of the $i$ elemental Gaussians (see Fig 4). Each Gaussian pulse identified in this way was considered an individual discrete control action. We defined the onset of each control action as the point in time defined by $\mu_i - 2\sigma_i$.

We computed cued reaction time (cued RT) as the time difference between the time of the GO cue and the onset of the first Gaussian power pulse. We computed the initial target capture direction by creating a vector between the home position and the hand's position at the time of the 1st Gaussian power peak (usually ~150ms into the movement). The average target capture time (TCT) was computed as the difference between the
onset of the first Gaussian power pulse and the moment of target capture for trials with hand power profiles best fit with a single Gaussian pulse. We computed *target capture direction error* as the difference between the initial target capture direction and the *intended target capture direction*, which was defined as vector between the home position and the target's position at the average target capture time (410 ms) during unperturbed conditions. For trials with hand power profiles best fit with more than one Gaussian pulse, *error correction latency* (ECL) was calculated as the time between the onset of the first Gaussian pulse and the onset of the second Gaussian pulse. We only considered the first two Gaussian pulses in our timing analyses, even though successful target capture often required more than one error correction.

![Figure 4](image.png)

Figure 4: Graphic depiction of Gaussian model fitting to hand power profiles. Black lines: the subject’s hand power time series; red lines: single Gaussian pulses (submovements); black dotted line: the sum of Gaussians; blue lines: the second derivative of hand power (used to determine how many Gaussian pulses to include in the optimization). **A)** a trial that successfully captured the target with a single primary movement. The second derivative of hand power shows 1 zero crossing (small black arrow) indicating that only 1 Gaussian should be used to fit the data. **B)** a trial with 3 zero crossings in the second derivative, indicating that the model should have 2 Gaussian pulses (see Appendix A for details on the Gaussian function fitting algorithm).
Statistical Hypothesis Testing

This study evaluated the timing and magnitude of error corrections that arise during a manual target interception task involving perturbations of visual target motion or perturbations of the hand's mechanical environment. We sought to test two hypotheses. First, we tested the extent to which arm movement error corrections during manual target interception reflect compensations for predicted performance errors rather than responses to actual performance errors (FishBach et al. 2007). To do so, we used one-sided, one-sample t-test to compare the temporal difference between error correction latency and average target capture time to the null hypothesis value of zero (i.e., no difference). We expect the null hypothesis to be rejected if the time difference is negative, indicating that error corrections arise well before the average time of target capture. Second, we tested the hypothesis that the magnitude and timing of mid-movement error corrections reflect differences in the neural information processing required to infer the causal source of pending performance errors (whether they are self-, target-, or environment-generated) and to compensate appropriately for those errors. In this case, we applied repeated measures ANCOVA and post-hoc Dunnett’s t-test. In doing this we compared the dependent measures in the control condition to each one of the perturbation conditions individually with initial movement direction as a covariant. All data processing and model fitting were done in MATLAB 2019a (The MathWorks, Natick, Massachusetts).
All statistical analyses were conducted using the Minitab statistics software package (Minitab, LLC, State College, Pennsylvania). Statistical significance was set at a family-wise error rate of $\alpha = 0.05$.

3.3 Results

Overview of experiment

In this study we investigated error correction behaviors during a manual interception task involving perturbations of visual target motion or perturbations of the hand's mechanical environment. All fourteen compliant subjects were attentive throughout the experiments; each subject performed target capture movements with low movement onset latencies relative to the predictable GO cue. Across the study cohort and across all trial types, average cued RT values ranged from -150 ms to 251 ms. The average cued RT across all subjects (74 ± 108 ms) was well below the typical choice reaction time latencies (~300 ms; Mrotek 2013) that one would expect if participants had ignored the preparatory audio cues and only reacted to the final go cue. By providing a well-timed and predictable GO cue, and through judicious choice of target trajectories, we were able to control the initial direction of hand movement so that they effectively spanned the whole range of reach directions typically sampled during center-out reaching studies (Fig 5A). Across subjects, the realized initial movement directions closely approximated the intended directions (Fig 5B). Due to variations across the different target trajectories, subjects
made initial reaches that varied systematically in extent across movement directions (Fig 5C). For this reason, we included initial movement direction as a covariate in all subsequent analyses of variance reported below.

**Successful First Attempts**

A successful trial was one where the subject captured the target with only a single primary movement (i.e., a successful first attempt; see Fig 4A). Even though participants could anticipate the GO cue, the task was difficult; only 34±14% of unperturbed targets were caught on the first attempt. The task was similar in difficulty for the speed decrease condition, where first attempt success rate was 28±11%, and the viscosity increase conditions (38±12%). Performance was worse in both the speed increase
and viscosity increase conditions (22±13%) compared to the unperturbed condition (ANCOVA: F(4,681)= 15.08; p<0.0005). We analyzed trials wherein the subject acquired the target with only a single (primary) Gaussian hand power exertion to identify general characteristics of successful first-attempt target capture actions.

Because subjects were not informed in any way of whether a given trial was to be perturbed or unperturbed, it is unsurprising that they used a consistent strategy to initiate target capture actions regardless of trial type. The time of movement onset after the GO cue (cued RT) did not vary systematically across all unperturbed and perturbed trial types (ANCOVA: F(4,491)= 0.74; p=0.57; Fig 6A). Across subjects, the overall average cued RT was 78 ± 105ms; upon pooling all successful cued RTs across subjects, we find that the distribution is unimodal. During successful first attempt trials, we also observed significant differences in peak hand power across trial types (ANCOVA: F(4,491)= 3.29; p=0.011), however the planned Dunnett t-tests found no significant differences between any perturbation conditions and the unperturbed control condition (Fig 6B). Across subjects, peak hand power averaged 14.7±7.2 Nm/s. Considering peak hand power on all successful first attempt trials, we find that the distribution is again unimodal.
Similarly to hand power, hand speed magnitudes resulted in similar peak hand speeds for unperturbed and target speed perturbation trials (unperturbed: 0.55±0.14 m/s; speed increment: 0.59±0.17 m/s; speed decrement: 0.56±0.13 m/s), changing the nominal hand load caused substantial changes in peak hand speed (ANCOVA: $F(4,491.1)=36.4$; $p<0.0005$; Fig 7A). Peak hand speed increased notably when hand viscosity decreased (0.65 ±0.17 m/s), whereas peak hand speed decreased when viscosity increased (0.48±0.12 m/s). These hand speed differences caused differences in target capture times (ANCOVA: $F(4,491)=19.19$; $p<0.0005$; Fig 7B). The average target capture time (TCT) for successful first attempt unperturbed trials was 410 ± 89 ms. As expected, TCT varied by type of perturbation: successful first attempt speed perturbation trials required a similar amount of time to capture the target.
(speed decrease $422 \pm 94$ ms, and speed increase $411 \pm 102$ ms).

However, trials with a viscosity decrease took less time to capture the target ($373 \pm 94$ ms), and trials with a viscosity increase took longer to capture the target ($458 \pm 114$ ms).

**Fig. 7:** Cohort results of dependent performance variables across the five conditions (Trial Types). A) Peak hand speed; B) Target capture time (TCT). Mean values (+/- SEM) are shown for each condition type. Gray bar depicts one SEM above and below the mean for the control condition for comparison.

**Error Corrections**

Error correction (EC) trials were those where hand power profiles were best fit by a superposition of 2 or more elemental Gaussian pulses. The average number of hand submovements on trials with at least 1 error correction (i.e. at least 2 Gaussian pulses) was $4.0\pm0.4$; The number of required submovements was relatively uniform across all perturbation types except speed increases (ANCOVA: $F_{(4,673)}=44.43$; $p<0.0005$; Fig 8A). During speed increases the average number of submovements
increased from control to 6.2 ±0.7, presumably because attempting to catch a faster-moving target placed additional demands on the mechanisms predicting target motion (i.e., it was noticeably harder).

Fig. 8: Cohort results: initial actions on unsuccessful first attempt trials. A) Number of Submovements to intercept the target vs. Trial Type. Number of submovements was determined using Gaussian fits (see text). B) Cued Reaction Time vs. trial type. Grey dotted line depicts average reaction time for reaching movements without preparatory cues (Mrotek 2013); C) Peak hand Power vs. trial type. Mean values (+/- SEM) are shown for each condition type. Note in panel A that the error bars are smaller than the symbols themselves. For B & C, Gray bar depicts one SEM above and below the mean for the control condition for comparison.

We next analyzed cued RT, peak hand power, and target capture direction error for the (failed) primary target capture attempts on trials involving error corrections. We did so to identify general characteristics of unsuccessful first-attempt target capture actions (i.e., to answer the question: "What made unsuccessful target capture actions unsuccessful?").

The analysis showed that cued RT was almost identical between successful first attempt trials and trials with ECs (ANCOVA F(1,257)=0.17 P=0.916), indicating that this was not a factor influencing success or
failure in our experiment. On the other hand, we did find significant differences in the average hand power of the first movement between successful trials and EC trials (ANCOVA $F(1,257)=65.10$ $p<0.0005$). On average the hand power on successful first attempt trials was $14.7 \pm 7.2$ Nm/s while trials with error corrections had an initial hand power average of $11.48 \pm 7.24$ Nm/s. Here, the distribution of first movement power peaks is bi-modal. Finally, we looked at the target capture direction error and found significant differences on this as well (ANCOVA $F(1,257)=5.79$ $p=0.017$). The target capture direction error could either be positive, meaning that the subject aimed ahead of where the target would be at the average TCT (leading the target); or it could be negative, meaning that the subject aimed behind where the target would be at the average TCT (lagging the target). On average, trials with error corrections tended to lead the target more ($7.1 \pm 8.2$ Deg) than first attempt successful trials ($3.3 \pm 7.2$ Deg). These results indicate that both initial hand power and error correction angle were factors influencing whether or not the initial capture effort resulted in success or failure.

Given that the initial target capture efforts could fail for multiple reasons, mid-trial error corrections were frequently required to ultimately capture the moving target. We used Gaussian effort modeling to identify discrete actions that contributed to target capture performance. We computed the error correction latency as the time between the onset of the first Gaussian hand power pulse (i.e., the primary target capture effort)
and the onset of the second Gaussian pulse (i.e., the first error correction effort). We used repeated measures ANCOVA and Dunnett tests to test the hypothesis that haptic and visual perturbations differentially affect the timing and magnitude of error corrections relative to those made in the control condition. The analysis revealed a significant main effect of condition (ANCOVA $F_{(4,673)}=33.55$, $p<0.0005$; Fig 9A). Post-hoc Dunnett t-tests found that when compared to control trials, error correction latencies differed for hand viscosity perturbations (haptic) but not for target speed perturbations (visual). During control trials, subjects initiated an error correction $215 \pm 41$ ms after movement onset on average. By contrast, trials with a hand viscosity decrease had an initial error correction instigated with a longer latency ($272 \pm 44$ ms) and trials with a viscosity increase had an initial correction instigated with a shorter latency ($154 \pm 47$ ms). Initial error corrections in trials with target speed perturbations had similar latencies as unperturbed control trials (speed decreases: $222 \pm 41$ ms; speed increase: $233 \pm 52.3$ ms).
Fig 9: Characteristics of initial error corrections (cohort results)  

A) Error Correction Latency vs. Trial Type. Error bars: ± 1SEM. Horizontal grey band shows the Average Target Capture Time values (± 1 SEM) in successful first attempt trials. Note that in all conditions the Error Correction begins before the average capture time. 

B) Peak hand Power vs. Trial Type for the primary target capture effort (circles) and the initial error correction (squares). Mean values (+/- SEM) are shown for each condition type. 

Note that error correction latencies in all trial conditions (ranging from 154 ms to 272 ms on average) were far less than the average time of target capture in successful trials (Fig 9A, grey horizontal band). Initial error corrections begin before the subject would have reached the target if the primary effort had been successful, and well before subjects would have received sensory confirmation of a successful target capture. Subjects must therefore have initiated error correction actions based on predictions that the primary target capture action would fail, rather than on sensory confirmation that the attempt had already failed. 

The initial error corrections were not stereotyped responses. Rather, they were well-tuned to the perturbation conditions presented in each trial. While subjects produced similar primary target capture efforts
across all trial types during trials that required error corrections (ANCOVA: F\((4,673)\)=2.19, p=0.07; Fig 9B, circles), the vigor of the first error correction efforts varied systematically by trial type (ANCOVA: F\((4,673)\)=18.34, p<0.0005; Fig 9B, squares). The variations in initial error correction vigor reflected appropriate compensations for the imposed visual and mechanical perturbations. For example, initial error corrections in unperturbed trials exhibited hand power peaks averaging 5.2±1.8 Nm/s, and these peak values decreased in trials where the target speed decreased (4.3±1.3 Nm/s) and in trials where the robot handle viscosity decreased (3.3±1.2 Nm/s). Average peak hand power increased for error corrections in response to target speed increases (6.8±2.0 Nm/s) and in response to handle viscosity increases (7.3±2.4 Nm/s). Taken together, these findings demonstrate that on average within the first 154 ms to 272 ms of target interceptions, subjects predict the outcome of their primary effort, and adjust the vigor of the impending correction to compensate for any unexpected perturbation that may have been introduced.

### 3.4 Discussion

*Summary of Findings*

Participants grasped the handle of a 2D planar robot and moved to intercept targets moving along a 2D sum-of-cosines paths on a flat screen after a predictable Go cue. At the same time as the Go cue, on some trials (~1/3) the target motion speed was abruptly altered while on different trials
(\sim 1/3) the mechanical environmental viscosity intensity was altered.

During the initial interceptive movement there were key characteristics of
the planning and execution of the target interception that differentiated
successful and unsuccessful trials: successful trials generally had a more
vigorous initial movement and had an initial direction closer to the
intended target capture direction.

When participants were not successful on the first attempt, they
were instructed to continue trying to intercept the target. Thus, this study
allowed us to evaluate differences in the timing and magnitude of error
corrections between visual and haptic perturbations to test the hypothesis
that adaptations to changing task conditions that occur mid-movement
reflect errors in predicted rather than actual performance.

We found that changes in the viscosity of the robot arm significantly
altered the timing of the error correction response when compared to
unperturbed conditions: During viscosity decrease conditions the
corrective response latency took significantly longer to begin when
compared to unperturbed conditions; on the other hand, viscosity
increases led to significantly shorter target error correction latencies. We
demonstrated this by fitting a Gaussian model to the power profile of the
subject’s hand movements. We also showed that these corrective
responses happen in most cases before the subject reached the target,
and that their magnitude adapts to the type of perturbation presented on
the trial. Subjects were able to quickly identify errors made, incorporate
environmental and visual changes, and use this information to usher an appropriate corrective response.

**Success and Failure on First Attempt.**

Our first goal was to determine "What made successful trials successful?". To do this we needed to find behavioral differences in successful and unsuccessful attempts. At first, we found several similarities between these attempts, indicating that these aspects of behavior did not differentiate success. We analyzed cued reaction time, and peak hand power and found that both did not differ between unperturbed and perturbed conditions, meaning that successful trials are initiated with the similar vigor and timing regardless of perturbation.

Similarly, we compared the cued reaction time, peak hand power and target capture direction error of the initial hand movement between successful in the first attempt trials and trials that needed more than one attempt to capture the target. We found that the cued reaction time was almost identical in both successful and trials with error corrections, which indicated that the timing of the initial movement was not a determining factor affecting success in our experiment.

However, there were aspects of behavior that differed, showing us specific aspects of the response that could predict a successful attempt. Peak power was significantly higher on first attempt successful trials than on error correction trials and target capture direction error was larger for
error correction trials than for first attempt successes (meaning that on average subjects aimed further ahead but used less power in their initial movement on trials with error corrections). At first glance these results seem contradictory, since using a strategy of aiming further ahead in the predicted target trajectory but performing a less vigorous movement would take the subjects longer to reach the target but should still allow them to successfully capture it on their first attempt. However, the success or failure of these two strategies make sense when we compare them with the two target interception strategies observed by Daum et al. (2007). In their study, Daum et al. found two types of target interception strategies based on whether the target was moving predictably or unpredictably. For predictable target movement the initial reaching movement direction was aimed further ahead towards where the target would be, but the movement speed was slower; while for unpredictable target movements the initial reaching movement was aimed closer to where the target was, but the movement speed was significantly faster. In our study the target would often change direction (enter a curve) soon after the go cue was given, introducing a level of unpredictability to the target movement. Therefore, an interception strategy of low power and high intended target capture direction error (akin to the predictable target movement strategy in Daum et al 2007) could not be an appropriate strategy in our experiment. In conclusion, results showed that a high power and low intended target capture direction error strategy was the one that would be most likely to
succeed at capturing the target on the first movement given that our target movement was designed to be unpredictable after the Go cue.

**Timing of error correction**

One of our main goals in this experiment was to determine if there are differences in error correction latencies after unexpected visual or haptic perturbations. Veerman et al. (2008) showed that perturbing different visual attributes, such as luminance, size, color etc. led to differences in error correction latencies. In our study we compared error correction during speed changes (visual) and error correction during viscosity perturbations (haptic). Our results indicated that error correction latencies in speed perturbation conditions were no different from control (unperturbed) conditions. On the other hand, we found that the viscosity perturbations had a significant impact on error correction latency. Our results show that on average error corrections caused by decreases in viscosity lead to error correction latency increases of ~50ms when compared to unperturbed and speed perturbations, whereas viscosity increases lead error correction latencies ~70ms shorter.

Regarding the shorter latency for the viscosity increase perturbation, these results resemble those obtained by Camponogara & Volcic (2019), who found that haptic perturbations detected by the contralateral hand triggered error corrections faster than those detected visually. They proposed that this could be due to the transmission speed for somatosensory and proprioceptive signals from the periphery to the
primary somatosensory area. Somatosensory signals are transmitted faster than visual signal from the retina to the primary visual cortex (Arnfred 2005; Mima et al. 1996; Walsh et al. 2005, Cluff et al. 2015). However, this would not address the stark differences for the long latencies seen during viscosity decreases. We propose that the long error correction latencies during decreases in viscosity are simply biomechanical in nature: slowing down the arm to perform a correction requiring a dramatic change in direction due to a decrease in viscosity takes longer to complete than increasing the force in an already active muscle to push harder after a viscosity increase. Further research using muscle activity as well as kinematic data to time the error correction latency, similar to studies conducted by (Debicki and Gribble 2004, and Kutzer, Pruszynski and Scott 2009) should be considered to fully answer this question.

The specificity of the magnitude of the corrective responses to altered target and environmental dynamics.

As a trial began and the subject prepared the motor plan to successfully capture the target, (s)he has no knowledge of whether a perturbation will happen and if so which type of perturbation it might be. It follows then that the subject prepared for the initial movement using the most current information presented, namely the speed of the target, its trajectory, angular velocity and the viscosity field most commonly applied to the robotic handle. Since the velocity and viscosity field are always kept
the same at the beginning of each trial, it is expected that the first movement magnitude is planned to be the same for unperturbed and perturbed conditions alike. As shown in Figure 6, this was the case in our experiment. All trial conditions had similar hand power for the first movement. However, the behavior did not remain consistent across conditions if the target was missed on the first attempt. Whether a target miss was due to a self-generated error, an error in the prediction of target motion in speed jump trials, or an error in prediction of the hand's load when the robot's handle viscosity changed, during the error correction the hand movement was adjusted to the environment quickly and skillfully.

Studies such as Wijdenes et al. (2011) have shown that the magnitude of a corrective response can vary depending on the time when the perturbation is presented. They used manual interception and target position perturbations at different times during the reaching movement and found that later perturbations caused more intense responses than early perturbations. Other studies have argued that both behavior and latency of the corrective response is affected by the timing of the perturbation (Mrotek 2013). In our study we have shown that the magnitude of the error correction response was also proportional to the type of perturbation and that this adaptation occurred in under 300 ms. For visual perturbations, an increase in speed elicited a more powerful response, in order to accelerate and successfully capture the target. The same is true for increases in viscosity: presented with a higher resistance to movement the
subject adapted the error correction response and vigorously added power to the corrective movement for a successful capture. On the other hand, decreases in target speed or viscosity intensity ushered a much more “low-powered” error correction response as the target had slowed down or the robotic handle now required less force to manipulate. We conclude therefore that these error corrections are not stereotyped responses but are fine tuned to the circumstances.

The role of prediction in the production of mid-movement error corrections

Finally, we were interested in learning whether arm movement error corrections generated during our experiment reflected compensations for predicted performance errors or whether they were in response to actual performance errors (FishBach et al. 2007). There is increasing evidence suggesting that visual (for a review see Brenner and Smeets 2018) and haptic (Flanagan et al. 2003) sensory information is continually being processed during manual interception tasks and used to make predictions of target and hand trajectory. When a discrepancy between the predicted path and the current hand movement is detected, we move to correct for the error immediately (Paulignan et al. 1991, Fink et al. 2009, Mrotek 2013, Brenner and Smeets 2016).

Even though error correction latencies in our experiment varied across error generation mechanisms, in all conditions the first error correction systematically began before the subject would have reached the target with their first initial movement and well before subjects would
have received sensory confirmation of a successful target capture. Therefore, the error correction must have been initiated in response to a prediction that the primary target capture effort would not be successful, supporting the idea that target and hand movement is continually being updated and used to make predictions of the interceptive movement. This predictive behavior observed in our experiment correlates closely with forward models of motor control such as the ones presented by Desmurget and Grafton (2000), Kawato (1999), Wolpert and Mail (1996).
Chapter 4: Limitations and Future Directions

There were several limitations to our experiment. We used the onset of the Go cue as the trigger for the perturbation. This method is valid as long as the subject follows instructions and uses the preparatory sound cues to time the initial movement, however as we saw in the results, the cued reaction time had a high variability, meaning that subject sometimes initiated movement too soon, which could change the latency or magnitude of the error correction in response to a perturbation (Wijdenes et al. 2011, Mrotek 2013), or too late, in which case the subject might be able to incorporate the visual perturbations into their initial movement strategy (Mrotek 2013). Future studies should consider matching the triggering of the perturbations to the onset of the subject’s movement or aborting the trial if the subject moves too soon or too late thereby eliminating onset of movement variability and maintaining a consistent initial movement direction.

Additionally it would be of great interest to analyze EEG data and EMG data during the experiment. EEG analysis would allow us to understand which regions of the brain are active during error corrections, and whether error corrections during the different perturbation conditions have different neural pathways. EMG data would lead to a better understanding of how the neuromuscular system responds to the different viscosity perturbations and could potentially allow us to better time the onset of the error correction by analyzing changes in muscle activity rather
than inferring its timing based on kinematic data. We could also examine the role of short latency responses such as reflexes in these types of situations.

Finally, the inclusion of a system such as EyeLink to analyze eye movement during our interception task would prove of great interest. It is well known that visual information plays a key role in manual interception, and smooth pursuit and saccadic data would allow us to describe the behavior of the eyes in response to visual perturbations (where there is a required change in eye movements) against haptic perturbations (where there is no change to the target motion) and investigate questions such as: what sort of mistakes were made by the eyes and whether they were similar to those made by the hand, how much the eyes lead the hand when trials are successful or unsuccessful, or how does eye movement differ between initial movement and subsequent error correction movements.
BIBLIOGRAPHY


Journal of Neurophysiology, 49(2), 548-564.  
Doi:10.1152/jn.1983.49.2.548. PMID: 6834087.


https://doi.org/10.1007/s00221-008-1296-x.

https://doi.org/10.1007/s00221-011-2843-4

https://doi.org/10.1016/s0893-6080(96)00035-4
APPENDICES

Appendix A: Preparatory Cue and Go Cue Arduino Code

/*
This Arduino code generates 3 consecutive low pitch sounds followed by 1 high pitch separated 333ms intervals (total cycle time 1s) using a MIDI shield. The sound acts as a preparatory Cue (3 low pitch sounds) and the Go Cue (final high pitch sound) for the study title "Mid-movement error corrections after visual and haptic perturbations reflect latency and performance differences".
This code has been designed as a state machine where that runs from one state to the next until it reaches the last state, after which the cycle resets. Additionally, if no signal is detected after a brief period (2s) the machine resets to State 1.
*/
#include <SoftwareSerial.h>
#include "Arduino.h"

SoftwareSerial mySerial(2, 3);

byte note = 0;
byte resetMIDI = 4;
byte ledPin = 13;
int instrument = 0;

int PitchPin = 7;
int CueEnablePin = 6;
int reset = 2000; //reset timer, milliseconds

static int Pitch, CueEnable, OldCueEnable=0, NoteTimer = 0;
static unsigned int state;
static unsigned long time;

void setup() {
    Serial.begin(57600);
    state = 1;
    time = 0;
    //Setup soft serial for MIDI control
    mySerial.begin(31250);

    //Reset the VS1053
    pinMode(resetMIDI, OUTPUT);
    digitalWrite(resetMIDI, LOW);
    delay(100);
    digitalWrite(resetMIDI, HIGH);
    delay(100);
talkMIDI(0xB0, 0x07, 120); //0xB0 is channel message, set channel volume to near max (127)

// setup inputs
pinMode(PitchPin, INPUT); //input to select sound pitch (not used in this iteration)
pinMode(CueEnablePin, INPUT); //input to switch on/off the sound cue

void loop() {

    Pitch = digitalRead(PitchPin);
    CueEnable = digitalRead(CueEnablePin);
    // Serial.println(CueEnablePin);

    switch (state) {
    
    case 1:

        if (OldCueEnable == 0) {
            if (CueEnable == 1) {
                time = millis();
                Serial.println(time); // save time at which this state is triggered
                noteOn(0, 60, 127); // Low pitch sound
                NoteTimer = 100; // How long the sound is played
                state = 2;
            }
        }
        break;
    
    case 2:

        if (OldCueEnable == 0) {
            if (CueEnable == 1) {
                time = millis();
                Serial.println(time);
                noteOn(0, 60, 127);
                NoteTimer = 100;
                state = 3;
            }
            if (millis() - time > reset) // check if time passed
                state = 1;
        }
        break;

    case 3:

        if (OldCueEnable == 0) {
            if (CueEnable == 1) {
                time = millis();
                Serial.println(time);
                noteOn(0, 60, 127);
                NoteTimer = 100;
            }
        }
    }
state = 4;
}
if( millis() - time > reset) //check if time passed
    state = 1;
}
break;
case 4:
    if(OldCueEnable == 0){
        if (CueEnable == 1) {
            noteOn(0, 72, 127); //High pitch sound
            NoteTimer == 100;
            state = 1;
        }
        if( millis() - time > reset) //check if reset
            state = 1;
    }
break;
}

if (NoteTimer == 1)
{
    talkMIDI( (0xB0 | 0), 120, 0); // all notes off command
}

OldCueEnable = CueEnable;
NoteTimer = NoteTimer-1;
delay (1);
Appendix B: Robot Arm and Screen Display Code

Main.m
The entry point of the robot experiment. Main will call "compile"
which will compile TargetMU to the target and launch the simulink models.

clear % don't use 'clear all' - or else debug wont work
close all

% If you want to compile code to the target. Do not run 'compile.m'
% manually:
COMPILE = true;
% Rapid Experiment runs RapidExp.xls
DEBUG_RAPID_EXPERIMENT = false;

% Global List
global UDPsp UDPsps
global tg % Target computer controller object
global wxpc % Set in Main.m and used in compile.m
global Disp % Structure object of graphical handles of the stimulus
global sp % Sample Time used in SubWatchDog in TargetMU.mdl. Samp rate of 1000Hz
for scopes also.
global HostState % State of the host within a mode
global PracticeTrials % Number of practice trials
global Trial % Saved data structure of model scopes
global Comp % Flag if matlab should compile in compile.m
global Mode PrevMode % host state machine Mode counter (highest level of SM)
global NumSamplesPerMode % Defined in the TargetMU model parameters
global UDPBuffer % Variable that hold real-time variables for the Host. Populated elsewhere.
global WriteEnable % Enables the usage of the Host SM. Set in Main.m
global NowWriting oldTargetCount % Used in the Host_S_Fun_udp.m
global Home_X Home_Y % Starting home position of the robot handle.
global OpBtn OpBtnType
global Xec Yec
global LineDrawingX LineDrawingY LineDrawingIndex

% Home position of handle. These must match definitions in Forces_S
Home_X = -0.1400; % Global
Home_Y = 0.6425 - 0.05; % Global

% Create variables for debugging the line drawings
LineDrawingX = Home_X*ones(1,400);
LineDrawingY = Home_Y*ones(1,400);
LineDrawingIndex = 0;

% Mode Defs
MODE_CONCUSSION = 30;
MODE_BEST_EFFORT = 31;
MODE_FALSIFICATION = 32;
MODE_BEST_EFFORT_ASMT = 33;
MODE_FALSIFICATION_ASMT = 34;
MODE_ERRORCORRECTION_DEV = 40;
MODE_ERRORCORRECTION = 50;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Participant Characteristics
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The trial to start the experiment (can be changed if exp crashes and you
% want to resume)
FirstTrial = 154;

Trial.ExamineeData.Subj_Identifier = 'HR-1674.S001.S1'; % HR[IRB#].S[SubID].S[Session]
Trial.ExamineeData.DOB = '12/31/96'; % MM/DD/YY format
Trial.ExamineeData.Sex = 'F'; % M or F
Trial.ExamineeData.Hand_Used = 'R'; % Always use right hand for experiment
Trial.ExamineeData.ProtocolNum = MODE_ERRORCORRECTION; % 31 if Best Effort, 32 if Falsification, 33 if Best Effort ASMT, 34 if Falsification Assessment
Trial.ExamineeData.OrigSavePath = ['D:\Pablo\Data\HRxxxx\ErrorCorrectionExp\',Trial.ExamineeData.Subj_Identifier,'\'];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Trial.TargCount = -FirstTrial; % Negative values mean start points, otherwise positives begin at 1
% To simplify experimentation for non-programmers
switch Trial.ExamineeData.ProtocolNum
    case 31
        TestModes = MODE_BEST_EFFORT;
    case 32
        TestModes = MODE_FALSIFICATION;
    case 33
        TestModes = MODE_BEST_EFFORT_ASMT;
    case 34
        TestModes = MODE_FALSIFICATION_ASMT;
    case 40
        TestModes = MODE_ERRORCORRECTION_DEV;
    case 50
        TestModes = MODE_ERRORCORRECTION;
    otherwise
        error('Unrecognised Protocol Number. 31 if Best Effort, 32 if Falsification, 33 if Best Effort ASMT, or 34 if Falsification ASMT.');
end

Trial.ModeList = [1 TestModes]; % this is for Main experiment
Trial.AssessedMovementExtent = [NaN NaN]; % Initialize assessment value
Trial.HomePosition = [Home_X Home_Y]; % Save the home position

Trial.TargetPosition = [Home_X Home_Y-0.100]; % 10 cm away from subject's home

writeEnable = 1; % Used for UDP transfer
oldTargetCount = 0; % Used by UDP
OpBtn = false;
OpBtnType = 0;

Debug_SIM = true; % establish the sampling time of the scopes. Debug flag for compiling.
sp = 0.001; % Sample time of the Target. sp=0.001*25
UDPsp = sp*25; % communication udp (sample period of udp) 25ms = 40Hz
UDPsp2 = 1/UDPsp; % Samples per second of host (40 Hz)
Comp = true; % compile or not the c code. Used in compile.m
Wxpc = true;

load MU_Calib.mat % Loads load cell ACal and FCal to workspace for simulink load cell calibration.

Trial.ModeCount = 1;
Mode = Trial.ModeList(1); % Initializing the first value of Mode
PrevMode = 0; % Initialize the previous mode (none)

% Load the experiment parameters
if DEBUG_RAPID_EXPERIMENT % Run the rapid version of the experiment for debugging
else
    % Run the full version of the experiment
end

Disp = GenExpFig('H',false); % Generate the stimulus figure
set([Disp.Hfig Disp.Haxes], 'buttondownfcn', 'OperatorPressedButton') % Apply a button down function to the figure

Trial.VisualCalibrations = Disp.VisualCalibrations; % Copy the visual calibrations used for this experiment.

Disp = PlotScreen2(Disp); % Add experiment graphics

% Compile the experiment if desired
if COMPILE; compile; end

set(0,'currentfigure',Disp.Hfig) % Bring stimulus figure back in focus

function [sys,x0,str,ts] = Host_S_Fun(t,x,u,flag)

Main Host Function
Follows S Function protocol. Contains its own helper functions
switch flag
    case 0
        [sys,x0,str,ts] =mdlInitializeSizes;
        SetBlockCallbacks(gcbh); % gcbh handel to the current block
    case 2
        sys =mdlUpdate(t,x,u);
    case 3
        sys =mdlOutputs(t,x,u);
    case 'Start'
        LocalBlockStartFcn
    case { 1,4,9 'Stop'}
        sys =[];
    otherwise
        error(['Unhandled flag = ',num2str(flag)]);
end

Helper Function Modules

% Return the sizes, initial conditions, and sample times for the S-function.
function [sys,x0,str,ts] =mdlInitializeSizes()
    global UDPsp; % Sample rate of UDP

    sizes = simsizes;
    sizes.NumContStates  = 0;
    sizes.NumDiscStates  = 0;
    sizes.NumOutputs     = 0; % not dynamically sized
    sizes.NumInputs      = 0; % dynamically sized
    sizes.DirFeedthrough = 1; % has direct feedthrough
    sizes.NumSampleTimes = 1;

    sys = simsizes(sizes);
    str = [];
    x0  = [];
    ts  = [UDPsp 0]; % RAS 12/1/09 change to slow down the host.
end

% Return the output vector for the S-function
function sys =mdlOutputs(t,x,u)

    %Globals (A lot of these are not used)
    %Globals can exist in simulink!
    global tg % Target Object
    global UDPsp % Samples Per Second of Host (defined in main)
    global Disp % Graphics structure
    global Trial % Trial data structure for saving
    global Mode PrevMode HostState % State Machine codes
    % global Enable_Error_Sound_Flag
    global Home_X Home_Y % Reaching home position (defined in Main)
    global OpBtn OpBtnType % Host Operator Button
    global UDPBuffer NowWriting WriteEnable % UDP
    global StateTimer StateDelay % Host timing counters
global NextTrajectory
global LineDrawingX LineDrawingY LineDrawingIndex
global SendTargetParametersFlag
global GoCueState
global CurrentStair_GoStopTime DidHandMove ImplementStaircase

MaxLineDrawingBuffer = 400;

DEFAULT_GOSTOP_TIME = 450; % [ms] for staircase
STAIR_MOVEMENT_THRESHOLD = 0.01; % 1 cm radius circular window

if (writeEnable==0) % data is ready to be read. If not, do nothing
  % The UDPBuffer is loaded in Host_S-Fun_udp.m which is in
  UDPReceive.mdl which receives data from UDP_Data Transfer in TargetMU
  XPos = UDPBuffer.Xpos; % X-position of the manipulandum
  YPos = UDPBuffer.Ypos; % Y-position of the manipulandum
  GoCueState = UDPBuffer.SoundEnable;
  TargetState = UDPBuffer.TargetState; % The current state of the target
  TargetTime = UDPBuffer.Count_Time; % Timestamp of the target
  %StopCueState = UDPBuffer.ButtonType; % 1 = STOP!; 0 = ---
  ColorGoCueState = UDPBuffer.ButtonType;
  MaxVelocity = UDPBuffer.MaxVelocity; % Maximum velocity during reach
  XPosVision = UDPBuffer.XPosVision; % Target x-position from the target
  YPosVision = UDPBuffer.YPosVision; % Target y-position from the target
  % FlashScreenState = UDPBuffer.Flag_Sound_Vision;
  LineDrawingX (LineDrawingIndex+1) = XPosVision;
  LineDrawingY (LineDrawingIndex+1) = YPosVision;
  LineDrawingIndex = LineDrawingIndex +1;
  LineDrawingIndex = mod(LineDrawingIndex, MaxLineDrawingBuffer);

  writeEnable = 1; % notify the UDP that it can read the data to the
  buffer once again.

  HostState = int8(HostState); % Convert to 8-bit signed integer

  %UDPsp = 40; % samples per second of host
  WAIT_250_MSEC = 0.25*UDPsp; % Definition of 250ms

% Define Modes
MODE_INITIALIZE_PAUSE = 1; % Entry mode for all experiments
  MODE_CONCUSSION = 30;
  MODE_BEST_EFFORT = 31;
  MODE_FALSIFICATION = 32;
  MODE_BEST_EFFORT_ASMT = 33;
  MODE_FALSIFICATION_ASMT = 34;
  MODE_ERRORCORRECTION_DEV = 40;
  MODE_ERRORCORRECTION = 50;

% Define Target States used within modes. Must match target states.
  STATE_UNDEFINED = 0;
  STATE_COMPUTE_PATHS = 1;
STATE_DISPLAY_PATHS = 2;
STATE_SAFETY = 9;
STATE_SETUP_TRIAL = 10;
STATE_HOLD_AT_HOME = 11;
STATE_RAMP_FIELD = 12;
STATE_WAIT_HOST_GO = 13;
STATE_REACHING = 14;
STATE_REACH_DONE_DELAY = 15;
STATE_VEL_FB = 16;
STATE_ASSESS_ENDPOINT = 17;
STATE_TRIAL_DONE = 18;
STATE_EXPLORE = 19;
STATE_EC_LOAD_PARAMETERS = 20;
STATE_EC_WAIT_FOR_HOME = 21;
STATE_EC_WAIT_FOR_GO = 22;
STATE_EC_MOVING = 23;
STATE_EC_TARGET_CAPTURED = 24;
STATE_EC_TARGET_TIMEOUT = 25;
STATE_EC_TRIAL_HOLD = 26;
STATE_EC_TRIAL_DONE = 27;

% Define FReset Codes
% FReset = 1 is reserved.
F_HPERT_SENT = 2;
F_END_EXPLORE = 3;
F_HOST_GO = 4;
F_PARAMS_READY = 5;
F_HOST_ACK_BLACK_BTN = 7;
F_HOST_ACK_RED_BTN = 8;
F_HOST_ACK_ENTER_BTN = 9;
F_HOST_ACK_SYNC_REQUEST = 10;

% Define Host States
HOST_DEFAULT_STATE = 0;
HOST_SEND_PARAMETERS = 1;
HOST_SAVE_DATA = 8;
HOST_TARGET_DISPLAYING_PATH = 4;
HOST_MESSAGE_SCREEN = 9;
TRAJ_SET_A = 1;
TRAJ_SET_B = 2;
TRAJ_SET_C = 3;
TRAJ_SET_D = 4;
TRAJ_SET_E = 5;

switch Mode
  case MODE_INITIALIZE_PAUSE
    if PrevMode ~= MODE_INITIALIZE_PAUSE
set([Disp.hTarget Disp.hHome Disp.hCursor], 'visible', 'off');
setTarget(Mode, 'Mode'); % Send Mode to target
disp('MODE_INITIALIZE_PAUSE');
SendTargetParametersFlag = false;

TimeMove = inf;
PrevMode = MODE_INITIALIZE_PAUSE;
HostState = HOST_DEFAULT_STATE;
if Trial.TargCount <= 0
    Trial.TargCount = -Trial.TargCount; % TargCount from Main.m
else
    Trial.TargCount = 1;
end
end

switch HostState
    case HOST_DEFAULT_STATE
        Trial.ModeCount = Trial.ModeCount + 1; % Increment the Mode
        if Trial.ModeCount <= length(Trial.ModeList)
            % A new Mode is loaded
            set([Disp.hTarget Disp.hHome Disp.hCursor], 'visible', 'off');
            set(Disp.hStringLong, 'visible', 'on', 'string', ...
                ['Operator press to start mode ', num2str(Trial.ModeList(Trial.ModeCount))]);
            Originpos = get(Disp.Hfig, 'position');
            % Move mouse cursor to the figure.
            set(0, 'pointerLocation', Originpos(1:2)+[20 20])
            HostState = 1;
        else
            % The experiment is done
            set(Disp.hStringLong, 'visible', 'on', 'string', 'Experiment Finished')
            set([Disp.hBackground Disp.hTarget Disp.hCursor Disp.hHome], 'visible', 'off');
            Originpos = get(Disp.Hfig, 'position');
            set(0, 'pointerLocation', Originpos(1:2)+[20 20])
            stopProg % the count is longer than the list - stop program execution
    end
    case 1
        % Wait for Host button press
        if OpBtn
            OpBtn = false; % Force clear button flag (since we use OpBtn again very soon)
        end
        % Load the new Mode in the battery
Mode = Trial.ModeList(Trial.ModeCount);
HostState = 0;
set(Disp.hStringLong,'visible','off');
end
end
% End case MODE_INITIALIZER_PAUSE

case MODE_ERRORCORRECTION
    if PrevMode ~= MODE_ERRORCORRECTION
        disp('Setting Mode MODE_ERRORCORRECTION');
        setTarget(Mode,'Mode'); % Send Mode to target
        TimeMove = inf;
        PrevMode = MODE_ERRORCORRECTION;
        HostState = HOST_MESSAGE_SCREEN;
        CurrentStair_GoStopTime = DEFAULT_GOSTOP_TIME;
        ImplementStaircase = false;
        DidHandMove = false;
        SendTargetParametersFlag = false;
        StateTimer = 0;
        %                         NextTrajectory = TRAJ_SET_A;
        %                         if Trial.TargCount >= 0
        %                             Trial.TargCount = -Trial.TargCount; % TargCount from
        %                         else
        %                             Trial.TargCount = 1;
        %                         end
    end

    set(Disp.hTarget,'xdata', Home_X + XPosVision,'ydata', Home_Y + YPosVision) % Update green cursor coordinates
    set(Disp.hCursor,'xdata', XPos, 'ydata', YPos);

    %                         if TargetState == STATE_SAFETY
        % Safety limit was reached. Trial is
dead.
        %                         HostState = 9; % Skips to end of
        % current trial.
    %                         end

    switch HostState
        case HOST_MESSAGE_SCREEN
            %instructions code
            if (Trial.ExpParam(Trial.TargCount,1).TrialNum == 1)
                set(Disp.hCursor,'visible', 'off'); % Update green
cursor visibility
                set(Disp.hTarget,'MarkerFaceColor',
'y','MarkerEdgeColor', 'y','visible', 'off'); % manage the display of the go cue here
                set(Disp.hHome,'visible', 'of') % Display home
set(Disp.hStringLong,'visible','on','string',"Instructions: 1)Before trial begins go to Home Position.';... 
2)The target will appear after a couple seconds.';...
3)As the target is moving you will hear 4 beeps';... 
4)On the 4th beep the target will become blue';... 
5)you MUST attempt to capture it as soon as this happens';...
6)Trial ends when target is captured or 5s after color change.'));

set(Disp.hBackground, 'visible', 'off');
if OpBtn
  OpBtn = false; % Force clear button flag
(since we use OpBtn again very soon)
  % Load the new HostState in the battery
  HostState = HOST_DEFAULT_STATE;
  set(Disp.hStringLong,'visible','off');
end
%Take a break code
elseif (Trial.ExpParam(Trial.TargCount,1).TakeBreak == 1)
  set(Disp.hCursor,'visible', 'off'); % Update green
cursor visibility
  set(Disp.hTarget,'MarkerFaceColor', 'y','MarkerEdgeColor', 'y','visible', 'off'); % manage the display of the go cue here
  set(Disp.hHome,'visible', 'off') % Display home
cursor
  set(Disp.hStringLong,'visible', 'on','string','Rest Break');
  set(Disp.hBackground, 'visible', 'off');
  if OpBtn
    OpBtn = false; % Force clear button flag
(since we use OpBtn again very soon)
    % Load the new HostState in the battery
    HostState = HOST_DEFAULT_STATE;
    set(Disp.hStringLong,'visible','off');
  end
elseif (Trial.ExpParam(Trial.TargCount,1).TakeBreak == 2)
  set(Disp.hCursor,'visible', 'off'); % Update green
cursor visibility
  set(Disp.hTarget,'MarkerFaceColor', 'y','MarkerEdgeColor', 'y','visible', 'off'); % manage the display of the go cue here
  set(Disp.hHome,'visible', 'off') % Display home
cursor
  set(Disp.hStringLong,'visible','on','string',"Target now may speed or slow down

after go signal');
    set(Disp.hBackground, 'visible', 'off');
    if OpBtn
        OpBtn = false; % Force clear button flag
    end

    % Load the new HostState in the battery
    HostState = HOST_DEFAULT_STATE;
    set(Disp.hStringLong, 'visible', 'off');

    % Force clear button flag (since we use OpBtn again very soon)
    OpBtn = false; % Force clear button flag

    % Load the new HostState in the battery
    HostState = HOST_DEFAULT_STATE;
    set(Disp.hStringLong, 'visible', 'off');

    elseif (Trial.ExpParam(Trial.TargCount,1).TakeBreak == 3)
        set(Disp.hCursor, 'visible', 'off'); % Update green cursor visibility
        set(Disp.hTarget, 'MarkerFaceColor', 'y', 'MarkerEdgeColor', 'y', 'visible', 'off'); % manage the display of the go cue here
        set(Disp.hHome, 'visible', 'off') % Display home cursor
    end

    set(Disp.hStringLong, 'visible', 'on', 'string', {'Stop task begins:
      1)Capture target normally unless screen blinks red.
      2)If screen blinks red remain at the home position, DO NOT MOVE!' });
    set(Disp.hBackground, 'visible', 'off');
    if OpBtn
        OpBtn = false; % Force clear button flag
    end

    % Load the new HostState in the battery
    HostState = HOST_DEFAULT_STATE;
    set(Disp.hStringLong, 'visible', 'off');

    elseif (Trial.ExpParam(Trial.TargCount,1).TakeBreak == 3)
        set(Disp.hCursor, 'visible', 'off'); % Update green cursor visibility
        set(Disp.hTarget, 'MarkerFaceColor', 'y', 'MarkerEdgeColor', 'y', 'visible', 'off'); % manage the display of the go cue here
        set(Disp.hHome, 'visible', 'off') % Display home cursor
    end

    set(Disp.hStringLong, 'visible', 'on', 'string', {'Go to Home Target'});
    set(Disp.hBackground, 'visible', 'off');

    end

    case HOST_DEFAULT_STATE
    %
        if (StateTimer == 1)
            set(Disp.hCursor, 'visible', 'on'); % Update green cursor visibility
            set(Disp.hTarget, 'MarkerFaceColor', 'y', 'MarkerEdgeColor', 'y', 'visible', 'off'); % manage the display of the go cue here
            set(Disp.hHome, 'visible', 'on') % Display home cursor
            set(Disp.hStringLong, 'visible', 'on', 'string', 'Go to Home Target');
            set(Disp.hBackground, 'visible', 'off');
        end

        StateTimer = StateTimer + 1;

        if TargetState == STATE_EC_LOAD_PARAMETERS
HostState = HOST_SEND_PARAMETERS;
SendTargetParametersFlag = true;
DidHandMove = false;
StateTimer = 0;
end
if TargetState == STATE_EC_TRIAL_DONE
HostState = HOST_SAVE_DATA; % set a new target
trajectory
set(Disp.hCursor,'visible', 'off'); % update green
cursor visibility
set(Disp.hTarget,'visible', 'off');
set(Disp.hTarget,'markersize', 5.5*3); % manage
display of the go cue here
if (ImplementStaircase == true) % implement
staircase
    if (DidHandMove == false) % successful stop
trial
        CurrentStair_GoStopTime =
CurrentStair_GoStopTime + 50; % [ms] Therefore make it harder
    else
        CurrentStair_GoStopTime =
CurrentStair_GoStopTime - 50; % [ms] Therefore make it easier
        if CurrentStair_GoStopTime < 50
            CurrentStair_GoStopTime = 50;
        end
    end
end
StateTimer = 0;
end
if TargetState == STATE_EC_TRIAL_HOLD
set(Disp.hTarget,'markersize', 5.5*3*5); % manage
display of the go cue here
StateTimer = 0;
end
if TargetState == STATE_DISPLAY_PATHS
set(Disp.hTarget,'MarkerFaceColor', 'y', 'MarkerEdgeColor', 'y', 'visible', 'on'); % manage the display of the go cue here
HostState = HOST_TARGET_DISPLAYING_PATH;
set(Disp.hStringLong,'visible','off','string','Catch Target')
StateTimer = 0;
end

% if (StateTimer == 1)
% disp('In HostState 0.');
% end
% set(Disp.hLineDrawing,'xdata', Home_X +
LineDrawingX,'ydata', Home_Y + LineDrawingY, 'visible', 'on') % Update green
cursor coordinates
case HOST_TARGET_DISPLAYING_PATH %Target is displaying path, use this to control target display during trials.
    if (TargetState == STATE_DISPLAY_PATHS);
    set(Disp.hStringLong,'visible','off','string','Catch Target'); % don't do this every time through the loop
    if (StateTimer == 1)
        set(Disp.hCursor,'visible','on'); % Update cursor visibility
        set(Disp.hTarget,'visible', 'on');
        StateTimer = StateTimer + 1;
    end
    if (ColorGoCueState == 1)
        set(Disp.hTarget,'MarkerFaceColor', 'c','MarkerEdgeColor', 'c'); % manage the display of the go cue here
        disp('Setting Target Color');
    else
        set(Disp.hTarget,'MarkerFaceColor', 'y','MarkerEdgeColor', 'y'); % manage the display of the go cue here
    end
    if (Trial.ExpParam(Trial.TargCount,1).GoStopFlag == 1)
        if (StopCueState == 1)
            set(Disp.hBackground, 'visible', 'on');
        else
            set(Disp.hBackground, 'visible', 'off');
        end
    end
    if (DidHandMove == false)
        StairHandDisplacement = sqrt(((Home_X-XPos)*(Home_X-XPos))+((Home_Y-YPos)*(Home_Y-YPos)));
        if (StairHandDisplacement >= STAIR_MOVEMENT_THRESHOLD)
            disp(['*Hand disp: ' num2str(StairHandDisplacement)]); % manage the display of the go cue here
            DidHandMove = true;
        end
    else
        StateTimer = 0;
        HostState = HOST_DEFAULT_STATE;
    end

case HOST_SEND_PARAMETERS % set a new target trajectory
    if (SendTargetParametersFlag == true)
% manage GoStopTime staircase
if (Trial.ExpParam(Trial.TargCount,1).GoStopTime < -1)
    if (Trial.ExpParam(Trial.TargCount,1).GoStopFlag == 1)
        disp('Resetting staircase...');
    ImplementStaircase = true;
    else
        ImplementStaircase = false;
    end
elseif (Trial.ExpParam(Trial.TargCount,1).GoStopTime == -1)
    if (Trial.ExpParam(Trial.TargCount,1).GoStopFlag == 1)
        ImplementStaircase = true;
        disp('Following staircase...');
    else
        ImplementStaircase = false;
    end
else
    ImplementStaircase = false;
end

setTarget(Trial.ExpParam(Trial.TargCount,1).GoTime, 'GoCueTime');
setTarget(Trial.ExpParam(Trial.TargCount,1).SpeedJumpTime, 'TargetSpeedJumpTime');
setTarget(Trial.ExpParam(Trial.TargCount,1).SpeedJumpMultiplier, 'TargetSpeedMultiplier');
setTarget(Trial.ExpParam(Trial.TargCount,1).TargetPathRotation, 'TrajRotDeg');
setTarget(Trial.ExpParam(Trial.TargCount,1).TrialDuration, 'TrialTime');
setTarget(Trial.ExpParam(Trial.TargCount,1).GoStopTime, 'StopCueTime');
setTarget(Trial.ExpParam(Trial.TargCount,1).TimeMod, 'PathStart');
setTarget(Trial.ExpParam(Trial.TargCount,1).ViscosityMultiplier, 'TargetViscosityMultiplier');
    disp(['Go Time: ' num2str(Trial.ExpParam(Trial.TargCount,1).GoTime)]);
    disp(['Speed Jump Time: ' num2str(Trial.ExpParam(Trial.TargCount,1).SpeedJumpTime)]);
    disp(['Speed Jump Multiplier: ' num2str(Trial.ExpParam(Trial.TargCount,1).SpeedJumpMultiplier)]);}
disp(['Traj Time Mod [ms]: ' num2str(Trial.ExpParam(Trial.TargCount,1).TimeMod)]);
disp(['Target path: ' num2str(Trial.ExpParam(Trial.TargCount,1).TargetPath)]);
SendTargetParametersFlag = false;
end

TRAJ_SET_A)
if (Trial.ExpParam(Trial.TargCount,1).TargetPath == TRAJ_SET_A)
    setTarget(TRAJ_SET_A,'ParamSelect');
    setTarget(F_PARAMS_READY,'FReset');
    if (TargetState == STATE_EC_WAIT_FOR_GO)
        StartScope; % Start recording target data
        disp('acknowledge target response to load TRAJ_SET_A');
    end
    NextTrajectory = TRAJ_SET_B;
    HostState = 0;
    StateTimer = 0;
    setTarget(0,'FReset');
    disp('Exiting HostState 1A!...');
    %set(Disp.hLineDrawing,'visible', 'off') %
end

else
    %setTarget(0,'ParamSelect'); % reset
end

elseif (Trial.ExpParam(Trial.TargCount,1).TargetPath == TRAJ_SET_B)
    setTarget(TRAJ_SET_B,'ParamSelect');
    setTarget(F_PARAMS_READY,'FReset');
    if (TargetState == STATE_EC_WAIT_FOR_GO)
        StartScope; % Start recording target data
        disp('acknowledge target response to load TRAJ_SET_B');
    end
    NextTrajectory = TRAJ_SET_A;
    HostState = 0;
    StateTimer = 0;
    setTarget(0,'FReset'); % reset FReset
    disp('Exiting HostState 1B...');
    %set(Disp.hLineDrawing,'visible', 'off') %
end

else
    %setTarget(0,'ParamSelect'); % reset
end

elseif (Trial.ExpParam(Trial.TargCount,1).TargetPath == TRAJ_SET_C)
    setTarget(TRAJ_SET_C,'ParamSelect');
    setTarget(F_PARAMS_READY,'FReset');
    if (TargetState == STATE_EC_WAIT_FOR_GO)
StartScope; % Start recording target data
disp('acknowledge target respons to load

% reset FReset

NextTrajectory = TRAJ_SET_A;
HostState = 0;
StateTimer = 0;
setTarget(0,'FReset'); % reset FReset
disp('Exiting HostState 1C...');
%Set(Disp.hLineDrawing,'visible','off') %

Update green cursor coordinates
else
;
end
elseif (Trial.ExpParam(Trial.TargCount,1).TargetPath

== TRAJ_SET_D)

setTarget(TRAJ_SET_D,'ParamSelect');
setTarget(F_PARAMS_READY,'FReset');
if (TargetState == STATE_EC_WAIT_FOR_GO)
StartScope; % Start recording target data
disp('acknowledge target respons to load

% reset FReset

NextTrajectory = TRAJ_SET_A;
HostState = 0;
StateTimer = 0;
setTarget(0,'FReset'); % reset FReset
disp('Exiting HostState 1D...');
%Set(Disp.hLineDrawing,'visible','off') %

Update green cursor coordinates
else
;
end
else
setTarget(TRAJ_SET_E,'ParamSelect');
setTarget(F_PARAMS_READY,'FReset');
if (TargetState == STATE_EC_WAIT_FOR.GO)
StartScope; % Start recording target data
disp('acknowledge target respons to load

% reset FReset

NextTrajectory = TRAJ_SET_A;
HostState = 0;
StateTimer = 0;
setTarget(0,'FReset'); % reset FReset
disp('Exiting HostState 1..');
else
;
end
end % end if
(Trial.ExpParam(Trial.TargCount,1).TargetPath
case HOST_SAVE_DATA % safety limits reached
    set(Disp.hLineDrawing,'visible', 'off') % update
    green cursor coordinates
    set(Disp.hCursor,'visible', 'off'); % update green
    cursor visibility
    set(Disp.hTarget,'visible', 'off');
    setTarget(F_HOST_ACK_SYNC_REQUEST,'FReset');% set
    FReset to a sync request, which will let the target move on to request a new
    target trajectory
    if (TargetState == STATE_EC_WAIT_FOR_HOME)
        filer();
        Trial.TargCount = Trial.TargCount + 1; % Increment
    trial number
    if Trial.TargCount <= length(Trial.ExpParam) % If
    the end of the experiment is not reached
        set(Disp.hBackground, 'visible', 'on');
        HostState = HOST_MESSAGE_SCREEN; % Go to next
    trial
        else
        Mode = MODE_INITIALIZE_PAUSE; % experiment
    ended, goto next Mode
    end
    end

    case 9 % safety limits reached
    end % End case MODE_ERRORCORRECTION

    end % switch Mode

    end % End WriteEnable

    sys = []； % Return sys

    % Helper Functions
    function [EnterBtn, RedBtn, BlackBtn] = GetBtnFlags(BtnMask)
        % Converts button mask to individual button states

        % Button Bits
        % Matlab defines LSB as bit 1.
        BTN_BLACK = 1; % LSB
        BTN_RED = 2;
        BTN_ENTER = 3; % MSB

        % Decode Button Type
if bitget(BtnMask,BTN_BLACK)
    BlackBtn = true;
else
    BlackBtn = false;
end
if bitget(BtnMask,BTN_RED)
    RedBtn = true;
else
    RedBtn = false;
end
if bitget(BtnMask,BTN_ENTER)
    EnterBtn = true;
else
    EnterBtn = false;
end
end % End button helper function
end % End mdlOutputs

function sys = mdlUpdate(t,x,u)
    sys = [];
end

function SetBlockCallbacks(block)
end

function LocalBlockStartFcn()
end
end

Robot control code (written in the C programming language)
Follows S Function protocol.

/* File     : forces.c  'S-Function in C for XPC target'
 * Code: C  
 * Abstract: Moves the endpoint from current position to xo yo using an open loop
 * pd control
 *           *IMPORTANT* This function must reside inside an enable
 * subsystem configured to reset states.
 * * Input: Robot's endpoint position (X,Y) mm, initial position Xo,Yo mm,
 * and the four elements of the Jacobian
 * * Parameters:  Tt (sec) time to hold in initial position
 * * NV (mm/sec) nominal velocity. Velocity from current
 * position to initial position
 * * Output :  1 if Xo Yo reached and spent Tt sec in that position
 * 0 otherwise.
 */
#define S_FUNCTION_NAME Forces_S_Fun
#define S_FUNCTION_LEVEL 2
#include <math.h>
```c
#include "simstruc.h"
#define TRUE 1
#define FALSE 0
#define _PI 3.14159265358979

///////////////////////////////////////////////////////////////////////////
//  Jacobian Parameters
///////////////////////////////////////////////////////////////////////////
#define JACOBIAN_LEN 4 // Number of elements in the Jacobian
#define R1 0.460375 // Upper arm length in meters
#define R2 0.409575 // Lower arm length in meters

///////////////////////////////////////////////////////////////////////////
//  Safety Parameters
//  DO NOT EDIT! Manipulating these parameters can result in severe harm
//  to subjects and the robot. One motor can generate 44 lb-ft of torque
//  and can spin at 2150 RPM!
#define MAX_MOTOR_TORQUE 35 // Maximum torque that will be applied to the motors
#define MAX_POSITION_POS_X (0.110130000000000*2) // (Pos_Stop_LeftX) The left X limit of the table. Handle position must be greater than this value
#define MAX_POSITION_NEG_X (-0.423570000000000*1.5) // (Pos_Stop_RightX) The right X limit of the table. Handle position must be less than this value
#define MAX_POSITION_POS_Y (0.999000000000000*1.5) // (Pos_Stop_TopY) The bottom Y limit of the table. Handle position must be less than this value
#define MAX_POSITION_NEG_Y (-0.100000000000000*1.0) // (Pos_Stop_BottomY) The top Y limit of the table. Handle position must be greater than this value

#define SHOULDER_OFFSET -0.97357736558845
#define ELBOW_OFFSET -0.19012303544523
```
// Hand Position Offsets (meters)
#define X_OFFSET 0.01
#define Y_OFFSET -0.0025
///////////////////////////////////////////////////////////////////////////

// Mode Types
ckeditor ckeditor
#define MODE_INITIALIZE_PAUSE 1
#define MODE_CONCUSSION 30
#define MODE_BEST_EFFORT 31
#define MODE_FALSIFICATION 32
#define MODE_BEST_EFFORT_ASMT 33
#define MODE_FALSIFICATION_ASMT 34
#define MODE_ERRORCORRECTION_DEV 40
#define MODE_ERRORCORRECTION 50
///////////////////////////////////////////////////////////////////////////

// State Types
ckeditor ckeditor
#define STATE_UNDEFINED 0
#define STATE_COMPUTE_PATHS 1
#define STATE_DISPLAY_PATHS 2
#define STATE_SAFETY 9 // Enters this state if safety limitations are breached
#define STATE_SETUP_TRIAL 10
#define STATE_HOLD_AT_HOME 11
#define STATE_RAMP_FIELD 12
#define STATE_WAIT_HOST_GO 13
#define STATE_REACHING 14
#define STATE_REACH_DONE_DELAY 15
#define STATE_VEL_FB 16
#define STATE_ASSESS_ENDPOINT 17
#define STATE_TRIAL_DONE 18
#define STATE_EXPLORE 19
#define STATE_EC_LOAD_PARAMETERS 20
#define STATE_EC_WAIT_FOR_HOME 21
#define STATE_EC_WAIT_FOR_GO 22
#define STATE_EC_MOVING 23
#define STATE_EC_TARGET_CAPTURED 24
#define STATE_EC_TARGET_TIMEOUT 25
#define STATE_EC_TRIAL_HOLD 26
#define STATE_EC_TRIAL_DONE 27
#define TRAJ_SET_A 1
#define TRAJ_SET_B 2
#define TRAJ_SET_C 3
#define TRAJ_SET_D 4
#define TRAJ_SET_E 5
///////////////////////////////////////////////////////////////////////////

// FReset Codes
ckeditor ckeditor
#define F_HPERT_SENT 2
#define F_END_EXPLORE 3
#define F_HOST_GO 4
#define F_PARAMS_READY 5
#define F_HOST_ACK_BLACK_BTN 7
#define F_HOST_ACK_RED_BTN 8
#define F_HOST_ACK_ENTER_BTN 9
#define F_HOST_ACK_SYNC_REQUEST 10
///////////////////////////////////////////////////////////////////////////

// Button Codes
ckeditor ckeditor
// The button output is a flag bank (sum of bits) (supports multiple buttons at
```c
#define BTN_BLACK 1 // Binary 0000 0001
#define BTN_RED 2 // Binary 0000 0010
#define BTN_ENTER 4 // Binary 0000 0100

#define U(element) (*uPtrs[element]) /* Pointer to Input Port0 */
#define INPUTNO 21 // Number of inputs (U) into the Forces_S
#define OUTPUTNO 24 // Number of outputs (y) of Forces_S
#define SAMPLES_PER_SECOND 1000
#define NPARAMS 2

static int_T count = 1;
static int_T Enable_flag = 0;

/*====================*
* S-function methods *
*====================*/

/* Function: mdlInitializeSizes ===============================================
* Abstract:
*    The sizes information is used by Simulink to determine the S-function
*    block's characteristics (number of inputs, outputs, states, etc.).
*/
static void mdlInitializeSizes(SimStruct *S) {
    ssSetNumSFcnParams(S, NPARAMS); /* Number of expected parameters */
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 2); /* discrete states xdot, ydot */

    if (!ssSetNumInputPorts(S, 1)) return;
    ssSetInputPortWidth(S, 0, INPUTNO);
    ssSetInputPortDirectFeedThrough(S, 0, 1);

    if (!ssSetNumOutputPorts(S, 1)) return;
    //ssSetOutputPortWidth(S, 0, 2);      /* 2 outputs torque values */
    ssSetOutputPortWidth(S, 0, OUTPUTNO);

    ssSetNumSampleTimes(S, 1); // block based sample time
    ssSetNumWork(S, 0); // Specify the size of a block's floating-point work
    ssSetNumIWork(S, 0); // Specify the size of a block's integer work vector.
    ssSetNumDWork(S, 0); // Specify the size of a block's pointer work vector.
    ssSetNumModes(S, 0); // Specifies the size of the block's mode vector.
    ssSetNumNonsampledZCs(S, 0); // Specify the number of states for which a block
    // detects zero crossings that occur between sample points

    /* Take care when specifying exception free code - see sfuntmpl_doc.c */
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

/* Function: mdlInitializeSampleTimes =========================================
* Abstract:
*    Specifiy that we inherit our sample time from the driving block.
*/
static void mdlInitializeSampleTimes(SimStruct *S) {
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetoffSetTime(S, 0, 0.0);
}

/* Function: mdlOutputs ==============================================================
* Abstract:
*/
static void mdlOutputs(SimStruct *S, int_T tid) {
    real_T *y = ssGetOutputPortRealSignal(S, 0); // Get a pointer to
```
Debounce counters for buttons
in units of [ms]

Debounce counters for buttons

Debounce counters for buttons
Esort[MEDIAN_FILTER_LENGTH]:
static real_T TempS, TempE;
int_T LoopIndex, SortIndex;
static real_T Xa, Ya, RotX, RotY;
static real_T A1x, A2x, hx; Px, XScaleAbsMax, Shftx;
static real_T A1y, A2y, hy, Py, YScaleAbsMax, Shfty;
static real_T TrajPeriod, TrajSize, PhaseLoop, TimeScalar;
static real_T TrajRadIncrement, TrajRad;
static real_T HitBoxRadius;
static real_T RotMatrix[2][2] = {{1.0}, {0.1}};

UNUSED_ARG(tid); // not used in single tasking mode. Put this after variable declarations

///////////////////////////////////////////////////////////////////////////
// Assign S function inputs to the appropriate variables
///////////////////////////////////////////////////////////////////////////
ShIn0 = U(0); // Shoulder angle from encoder
ElIn0 = U(1); // Elbow angle from encoder
//PropCurve = U(2); // Curvature of the target path (from Host)
Hp = U(2); // Stiffness [N/m] of virtual spring applied to the HAND
Mode = (int_T) U(3); // stage of the experiment (from Host)
FReset = U(4); // FReset state sync (from Host)
FyIn0 = U(5); // Load cell force y. U(5) through U(10) are load cell forces and torques
FxIn0 = U(6); // Load cell force x. U(5) through U(10) are
iGoCueTime = (int_T) U(11); // [ms]
ITargetSpeedJumpTime = (int_T) U(12); // [ms]
ITargetSpeedJumpMul = (real_T) U(13); // Nominally 1.0
ParamSelect = (int_T) U(14); // Which set of tracking parameters to use?
TrajRotDeg = (int_T) U(15); // Rotation of Trajectory in Degrees
iStopCueTime = (int_T) U(16); // [ms]
Eyelink = (int_T) U(17); // not used in single tasking mode. Put this after variable
iPathStart = (int_T) U(18); // pick out each element in turn
iTargetViscosityMultiplier = (real_T) U(20); // Apply offsets to encoders (from calibrating them)
ShIn0 += SHOULDER_OFFSET;
ElIn0 += ELBOW_OFFSET;

///////////////////////////////////////////////////////////////////////////
// Implement Median Filter on Encoders
///////////////////////////////////////////////////////////////////////////
if (0) {
    for (LoopIndex=(MEDIAN_FILTER_LENGTH-1); LoopIndex>=1; LoopIndex--) {
        ShHist[LoopIndex-1] = ShHist[LoopIndex]; // keep the last MEDIAN_FILTER_LENGTH data values
        ShHist[LoopIndex] = ShIn0;
    }
    for (LoopIndex=0; LoopIndex<=MEDIAN_FILTER_LENGTH; LoopIndex++) {
        ShHist[LoopIndex] = ShHist[LoopIndex] + ShHist[LoopIndex-1]; // add the most recent data values to the history vectors
    }
}

// do "straight-insertion" sort (Numerical Recipes in C page 243)
// -- ascending numerical order
for (LoopIndex=1; LoopIndex<=MEDIAN_FILTER_LENGTH-1; LoopIndex++) {
    TempS = Ssort[LoopIndex]; // pick out each element in turn
    Ssort[LoopIndex] = Ssort[LoopIndex-1];
    while (SortIndex>=0 && Ssort[SortIndex]>TempS) { // look for the place to insert it
        Ssort[SortIndex+1] = Ssort[SortIndex];
        SortIndex--;
    }
    Ssort[SortIndex+1] = TempS; // insert it.
}

for (LoopIndex=1; LoopIndex<=MEDIAN_FILTER_LENGTH-1; LoopIndex++) {
    TempE = Esort[LoopIndex]; // pick out each element in turn
    Esort[LoopIndex] = Esort[LoopIndex-1];
    while (SortIndex>=0 && Esort[SortIndex]>TempE) { // look for the place to insert it
        Esort[SortIndex+1] = Esort[SortIndex];
        SortIndex--;
    }
    Esort[SortIndex+1] = TempE; // insert it.
}

for (LoopIndex=(MEDIAN_FILTER_LENGTH-1); LoopIndex>=1; LoopIndex--) {
    TempS = Ssort[LoopIndex]; // pick out each element in turn
    Ssort[LoopIndex] = Ssort[LoopIndex-1];
    while (SortIndex>=0 && Ssort[SortIndex]>TempS) { // look for the place to insert it
        Ssort[SortIndex+1] = Ssort[SortIndex];
        SortIndex--;
    }
    Ssort[SortIndex+1] = TempS; // insert it.
}

for (LoopIndex=0; LoopIndex<=MEDIAN_FILTER_LENGTH; LoopIndex++) {
    TempE = Esort[LoopIndex]; // pick out each element in turn
    Esort[LoopIndex] = Esort[LoopIndex-1];
    while (SortIndex>=0 && Esort[SortIndex]>TempE) { // look for the place to insert it
        Esort[SortIndex+1] = Esort[SortIndex];
        SortIndex--;
    }
    Esort[SortIndex+1] = TempE; // insert it.

}


```c
if (iButtonUp == 0) {
  // as a clean press.
  // Debouncing
}
else {
  // Load the history variables
  FXIn2 = FXin1; FXin1 = FXin0; FXOut2 = FXOut1; FXOut1 = FXOut0;
  FYIn2 = FYin1; FYin1 = FYin0; FYOut2 = FYOut1; FYOut1 = FYOut0;
  ShIn2 = Shin1; Shin1 = Shin0; ShOut2 = ShOut1; ShOut1 = ShOut0;
  FyIn2 = FyIn1; FyIn1 = FyIn0; FyOut2 = FyOut1; FyOut1 = FyOut0;
  ShIn2 = Shin1; Shin1 = Shin0; ShOut2 = ShOut1; ShOut1 = ShOut0;
  ElIn2 = ElIn1; ElIn1 = ElIn0; ElOut2 = ElOut1; ElOut1 = ElOut0;
} // if (count == 2)
else {
  // Generate the outputs
  FX = FXOut0 = B0_20Hz*FXin0 + B1_20Hz*FXin1 + A1_20Hz*FXOut1 -
  A1_20Hz*FXOut2 - A2_20Hz*FXOut1;
  FY = FYOut0 = B0_20Hz*FYin0 + B1_20Hz*FYin1 + A1_20Hz*FYOut1 -
  A1_20Hz*FYOut2 - A2_20Hz*FYOut1;
  ShAng = ShOut0 = B0_20Hz*Shin0 + B1_20Hz*Shin1 - A1_20Hz*ShOut1;
  ElAng = ElOut0 = B0_20Hz*ElIn0 + B1_20Hz*ElIn1 - A1_20Hz*ElOut1;
  // Load the history variables
  FXIn2 = FXin1; FXin1 = FXin0; FXOut2 = FXOut1; FXOut1 = FXOut0;
  FYIn2 = FYin1; FYin1 = FYin0; FYOut2 = FYOut1; FYOut1 = FYOut0;
  ShIn2 = Shin1; Shin1 = Shin0; ShOut2 = ShOut1; ShOut1 = ShOut0;
  FyIn2 = FyIn1; FyIn1 = FyIn0; FyOut2 = FyOut1; FyOut1 = FyOut0;
  ShIn2 = Shin1; Shin1 = Shin0; ShOut2 = ShOut1; ShOut1 = ShOut0;
  ElIn2 = ElIn1; ElIn1 = ElIn0; ElOut2 = ElOut1; ElOut1 = ElOut0;
} // if (count == 2)
else {
  // Generate the outputs
  FX = FXOut0 = B0_20Hz*FXin0 + B1_20Hz*FXin1 + A1_20Hz*FXOut1 -
  A1_20Hz*FXOut2 - A2_20Hz*FXOut1;
  FY = FYOut0 = B0_20Hz*FYin0 + B1_20Hz*FYin1 + A1_20Hz*FYOut1 -
  A1_20Hz*FYOut2 - A2_20Hz*FYOut1;
  ShAng = ShOut0 = B0_20Hz*Shin0 + B1_20Hz*Shin1 + B2_20Hz*ShIn2 -
  A1_20Hz*ShOut1 - A2_20Hz*ShOut2;
  ElAng = ElOut0 = B0_20Hz*ElIn0 + B1_20Hz*ElIn1 + B2_20Hz*ElIn2 -
  A1_20Hz*ElOut1 - A2_20Hz*ElOut2;
  // Load the history variables
  FXIn2 = FXin1; FXin1 = FXin0; FXOut2 = FXOut1; FXOut1 = FXOut0;
  FYIn2 = FYin1; FYin1 = FYin0; FYOut2 = FYOut1; FYOut1 = FYOut0;
  ShIn2 = Shin1; Shin1 = Shin0; ShOut2 = ShOut1; ShOut1 = ShOut0;
  FyIn2 = FyIn1; FyIn1 = FyIn0; FyOut2 = FyOut1; FyOut1 = FyOut0;
  ShIn2 = Shin1; Shin1 = Shin0; ShOut2 = ShOut1; ShOut1 = ShOut0;
  ElIn2 = ElIn1; ElIn1 = ElIn0; ElOut2 = ElOut1; ElOut1 = ElOut0;
} // else (count > 2)
```
} else if (iButtonUp == 1){
    // Physical button is not pressed
    BtnUpCtr = 0; // Reset black/up debounce counter
}

// Poll the Red/Down button
if (iButtonDn == 0){
    // Physical button is being pressed
    BtnDnCtr++; // Increment debounce timer
    if (BtnDnCtr >= DEBOUCE_TIME*SAMPLES_PER_SECOND){
        // We have a clean button press. Send button output.
        y[18] += BTN_RED; // Add Red button bit to output
        // Continue to send this output so long as
        // the button is pressed (or state change).
    }
} else if (iButtonDn == 1){
    // Physical button is not pressed
    BtnDnCtr = 0; // Reset Red/Down debounce counter
}

// Poll the Enter button
if (iButtonEnter == 0){
    // Physical button is being pressed
    BtnEnterCtr++; // Increment debounce timer
    if (BtnEnterCtr >= DEBOUCE_TIME*SAMPLES_PER_SECOND){
        // We have a clean button press. Send button output.
        y[18] += BTN_ENTER; // Add Enter bit to output
        // Continue to send this output so long as
        // the button is pressed (or state change).
    }
} else if (iButtonEnter == 1){
    // Physical button is not pressed
    BtnEnterCtr = 0; // Reset Enter debounce counter
}

///////////////////////////////////////////////////////////////////////////
*/

///////////////////////////////////////////////////////////////////////////
/**

Assign the default output values

Assign the default output values

/**

// Notice the position of y (i.e y[0], y[7], ...) is different from the
// tg.output column position in the output matrix which is defined according to
the simulink model

y[0] = y[1] = 0; // the Tx and Ty torques (often calculated from tempFx and
// tempy)
// y[2]=0;       // Sound Enable - don't reset this on every iteration!!!
// y[3]=0;       // Software Enable
// y[4]=0;       // Sound pitch selector (0 = low; 1 = high;
don't reset on every iteration! Also used for blinking screen
// y[5] = 0;     // State
// y[6] = 0;     // Iterations the Forces_S_fun has been run (if 1 kHz,
then each count is 1ms)
// y[7] = y[8] = 0; // Instantaneous forces in the x and y coordinates (tempFx,
tempy)

// Calculate the Jacobians using shoulder and elbow data
J[0] = -R1 * sin(ShAng); // J00
J[1] = R1 * cos(ShAng); // J10
J[2] = -R2 * sin(ElAng); // J01
J[3] = R2 * cos(ElAng); // J11

// Save previous position (for velocity calculation)
OldXh = xhS;
OldYh = yhS;

// Calculate new position
xhs = J[1]*J[3]; // Hand X position
yhS = -(J[0]*J[3]); // Hand Y position
// Add offsets to hand
xhs += X_OFFSET;
yhS += Y_OFFSET;

// Velocity (New - Old position)
vx = (xhs - OldXh) * SAMPLES_PER_SECOND;
vy = (yhS - OldYh) * SAMPLES_PER_SECOND;

y[9] = xhs; // "XPos_filtered"
y[10] = yhS; // "YPos_filtered"
y[11] = dLCCalibrationOut[0]; // Loadcell Offset values (not used)
y[12] = dLCCalibrationOut[1];
y[13] = dLCCalibrationOut[2];
```c
y[14] = dlCCalibrationOut[3];

y[15] = dlCCalibrationOut[4];

y[16] = dlCCalibrationOut[5];

y[17] = 0; // free

y[18] = 0; // free

y[19] = 0; // Maximum Velocity

y[20] = ShAng; // send the filtered angles outside of this c code

y[21] = ElAng;

y[22] = 0; // XPosVis

y[23] = 0; // YPosVis

// Default to no forces, unless requested otherwise

tempfx = 0;

tempfy = 0;

// Check if mode had changed (transition between modes)
if (PrevMode != Mode)
  PropExpState = STATE_UNDEFINED;

PrevMode = Mode;

// safety - check that robot does not move too fast
if (count == 1)
  y[3] = Enable_flag = 0;
else{
  if (count <= 20) // let the filters catch up.
    
  else // count > 20
    
    y[3] = Enable_flag; // Enable_flag; // set the output
}

switch (Mode) // Enter the Experiment State machine
{

  case MODE_INITIALIZE_PAUSE:
    y[0] = y[1] = 0; // zero output torques
    PropExpState = STATE_UNDEFINED;
    StateIndex = 0;
    break;

  case MODE_ERRORCORRECTION:
    y[0] = y[1] = 0; // zero output torques
    y[2] = 0; // disable sound output
    y[4] = 0; // set sound pitch low/ background off
    y[5] = PropExpState; // Send the current state

    switch (PropExpState)
    {
    case STATE_UNDEFINED: // 0 Entry Point
      tempfx = 0; tempfy = 0; // output zero torque
      StateIndex = 0;
      y[22] = -0.1; // XPosVis for debugging
      y[23] = -0.1; // YPosVis for debugging
      PropExpState = STATE_EC_WAIT_FOR_HOME; // Go to load
      break; // switch (PropExpState)
      
    case STATE_EC_LOAD_PARAMETERS:
      tempfx = 0; tempfy = 0; // output zero torque

      if (FReset == F_PARAMS_READY) 
        // wait until host signals the param variable is ready

      
    RotMatrix[0][0] = cos((real_T)(TrajRotDeg)/180.0+_PI);
    RotMatrix[0][1] = -sin((real_T)(TrajRotDeg)/180.0+_PI);
    RotMatrix[1][0] = sin((real_T)(TrajRotDeg)/180.0+_PI);
    RotMatrix[1][1] = cos((real_T)(TrajRotDeg)/180.0+_PI);

    GoCueTime = iGoCueTime; // [ms]
    StopCueTime = iStopCueTime; // [ms]
    TargetSpeedJumpTime = iTargetSpeedJumpTime; // [ms]
    SpeedJumpMul = 1.0; // multiplacitive factor
    TrialTime = iTrialTime; // [ms]
    PathStart = iPathStart; // [ms]
    
    iTargetViscosityMultiplier;
```
switch (ParamSelect) {
  case TRAJ_SET_A:
    A1x = 5.4; // [cm]
    A2x = 5.4; // [cm]
    hx = 3.0; // unitless
    Px = ((real_T)90/360)*2*3.14; // phase angle
    shiftx = 0; // this variable is used to shift path left/right so that target does not go through home
    A1y = 2.7; // [cm]
    A2y = 5.4; // [cm]
    hy = 2.0; // unitless
    Py = ((real_T)-25/360)*2*3.14; // phase angle [rad]

  case TRAJ_SET_B:
    A1x = 4.32; // [cm]
    A2x = 4.32; // [cm]
    hx = 3.0; // unitless
    Px = ((real_T)90/360)*2*3.14; // phase angle
    shiftx = 0;
    A1y = 2.7; // [cm]
    A2y = 4.32; // [cm]
    hy = 2.0; // unitless
    Py = ((real_T)50/360)*2.0*3.14; // phase angle [rad]

  case TRAJ_SET_C:
    A1x = 2.7; // [cm]
    A2x = 5.4; // [cm]
    hx = 2.0; // unitless
    Px = ((real_T)35/360)*2*3.14; // phase angle
    shiftx = 0;
    A1y = 4.32; // [cm]
    A2y = 4.86; // [cm]
    hy = 3.0; // unitless
    Py = ((real_T)260/360)*2.0*3.14; // phase angle [rad]

  case TRAJ_SET_D:
    A1x = 2.7; // [cm]
    A2x = 3.5; // [cm]
    hx = 3.0; // unitless
    Px = ((real_T)85/360)*2*3.14; // phase angle
    shiftx = 0;
    A1y = 2; // [cm]
    A2y = 5.3; // [cm]
    hy = 4.0; // unitless
    Py = ((real_T)300/360)*2.0*3.14; // phase angle [rad]

  case TRAJ_SET_E:
    A1x = 3.0; // [cm]
    A2x = 2.0; // [cm]
    hx = 3.0; // unitless
    Px = ((real_T)250/360)*2*3.14; // phase angle
    shiftx = 0;
    A1y = 3; // [cm]
    A2y = 3; // [cm]
    hy = 2.0; // unitless
    Py = ((real_T)250/360)*2*3.14; // phase angle

  case TRAJ_SET_F:
    A1x = 2.7; // [cm]
    A2x = 2.7; // [cm]
    hx = 3.0; // unitless
    Px = ((real_T)190/360)*2*3.14; // phase angle
    shiftx = 0;
    A1y = 4.86; // [cm]
    A2y = 4.86; // [cm]
    hy = 5.4; // unitless
    Py = ((real_T)350/360)*2.0*3.14; // phase angle

  case TRAJ_SET_G:
    A1x = 2.7; // [cm]
    A2x = 2.7; // [cm]
    hx = 3.0; // unitless
    Px = ((real_T)190/360)*2*3.14; // phase angle
    shiftx = 0;
    A1y = 4.86; // [cm]
    A2y = 4.86; // [cm]
    hy = 5.4; // unitless
    Py = ((real_T)350/360)*2.0*3.14; // phase angle

  default:
    break;
}
angle [rad]
shiftx = 0;
A1y = 3.0; // [cm]
A2y = 7.7; // [cm]
hy = 2.0; // unitless
Py = ((real_T)90/360)*2.0*3.14; // phase angle

angle [rad]
shifty = 0;
XScaleAbsMax = 4.48405399923099;
YScaleAbsMax = 9.8904856482679;
TrajSize = 0.15; /* [cm] */
break;
default:
A1x = 5.4; // [cm]
A2x = 5.4; // [cm]
hx = 5.0; // unitless
Px = ((real_T)35/360)*2*3.14; // phase angle

angle [rad]
shifty = 0;
XScaleAbsMax = 3.75765535550890;
YScaleAbsMax = 5.94306506514520;
TrajSize = 0.15; /* [cm] */
// add Go cue timing here
break;
}

// from switch (ParamSelect)
TimeScalar = 1000; /* 1000 */
TrajPeriod = 10;
//TrajRad = iPathStart;
TrajRad = (real_T)(((1.0)/TimeScalar)*iPathStart)^(2.0*PI)/TrajPeriod);
TrajRadIncrement = (real_T)(((1.0)/TimeScalar)^(2.0*PI)/TrajPeriod);
PropExpState = STATE_EC_WAIT_FOR_GO;
StateIndex = 1;
}
break; // switch (PropExpState)

case STATE_EC_WAIT_FOR_GO:
tempfx = 0; tempfy = 0; // output zero torque
if (FReset == F_HOST_ACK_SYNC_REQUEST){
    // wait until host resets the FReset signal
    PhaseLoop = 0;
    StateIndex = 0;
    PropExpState = STATE_DISPLAY_PATHS;
}
break;

case STATE_EC_WAIT_FOR_HOME:
tempfx = 0.2*ENVIRONMENTAL_VISCOSITY*(Vx);
tempfy = 0.2*ENVIRONMENTAL_VISCOSITY*(Vy);
if ((xhS>(Home_X + 0.003)) && (xhS>(Home_X - 0.003)) && (yhS>(Home_Y + 0.003)) && (yhS>(Home_Y - 0.003)))
    { StateIndex++; if (StateIndex >= 1500){
        StateIndex = 0; // Reset counter
        PropExpState = STATE_EC_LOAD_PARAMETERS;
    }
}
break;

case STATE_EC_MOVING:
tempfx = 0; tempfy = 0; // output zero torque
break;
case STATE_EC_TRIAL_DONE: // wait for host to indicate that it has saved the data
    tempfx = 0; tempfy = 0; // output zero torque
    if (FReset == F_HOST_ACK_SYNC_REQUEST){ // wait until host resets the FReset signal
        PhaseLoop = 0;
        StateIndex = 0;
        PropExpState = STATE_EC_WAIT_FOR_HOME;
    }
    break;

case STATE_EC_TRIAL_HOLD: // wait for host to indicate that it has saved the data
    tempfx = 0; tempfy = 0; // output zero torque
    if (FReset == F_HOST_ACK_SYNC_REQUEST){ // wait until host resets the FReset signal
        PhaseLoop = 0;
        StateIndex = 0;
        PropExpState = STATE_EC_WAIT_FOR_HOME;
    }
    break;

case STATE_DISPLAY_PATHS: // 2
    tempfx = iTargetViscosityMultiplier * ENVIRONMENTAL_VISCOSITY * (Vx);
    tempfy = iTargetViscosityMultiplier * ENVIRONMENTAL_VISCOSITY * (Vy);
    y[22] = RotX; // keep displaying old XPosTarget
    y[23] = RotY; // keep displaying old YPosTarget
    if (StateIndex++ >= 1000){ // wait at target for 1 second
        PhaseLoop = 0;
        StateIndex = 0;
        PropExpState = STATE_EC_TRIAL_DONE;
    }
    break;

if ((StateIndex + 999 >= iGoCueTime) && (StateIndex + 899 <= iGoCueTime)) // manage the Go Cue Feedback Signal (how handle pre-cues?)
    { y[2] = 1; // give Go Cue on signal 'y[2]' y[18] = 0; }
if ((StateIndex + 666 >= iGoCueTime) && (StateIndex + 566 <= iGoCueTime)) // manage the Go Cue Feedback Signal (how handle pre-cues?)
    { y[2] = 1; // give Go Cue on signal 'y[2]' y[18] = 0; }
if ((StateIndex + 333 >= iGoCueTime) && (StateIndex + 233 <= iGoCueTime)) // manage the Go Cue Feedback Signal (how handle pre-cues?)
    { y[2] = 1; // give Go Cue on signal 'y[2]' y[18] = 0; }
if ((StateIndex >= iGoCueTime) && (StateIndex <= iGoCueTime + 100))//
    { y[2] = 1; // give Go Cue on signal 'y[2]' }
if (StateIndex >= iGoCueTime)
    { y[18] = 1; // Color change of target }
    else
    { // y[2] = 0;
    // y[18] = 0;
    //}
    if (StateIndex >= (iGoCueTime + iTargetSpeedJumpTime))
    { SpeedJumpMul = iTargetSpeedJumpMul;
        TrajRadIncrement = (real_T)((real_T)TrajPeriod -
        SpeedJumpMul) / TimeScalar * (2.0 * PI) / TrajPeriod;
        tempfx =
TargetViscosityMultiplier*0.2*ENVIROMENTAL_VISCOSITY*(vx);

TargetViscosityMultiplier*0.2*ENVIROMENTAL_VISCOSITY*(vy);
}

// y[18] = TrajRadIncrement;
TrajRad = TrajRad + TrajRadIncrement;

if (1) {
Xa = 1*(TrajSize*(((A1x*cos(((real_T)TrajRad) +
A2x*cos(hx*(real_T)TrajRad - PX))/XScaleAbsMax) + Shiftx);
Ya = (TrajSize*(((A1y*cos(((real_T)TrajRad) +
A2y*cos(hy*(real_T)TrajRad - PY))/YScaleAbsMax)) + Shift;
}
else {
Xa = 1*TrajSize*(((A1x*cos(((real_T)StateIndex +
(real_T)PathStart)*SpeedJumpMul)/TimeScalar)*2.0*PI)/TrajPeriod +
A2x*cos(hx((((real_T)StateIndex +
(real_T)PathStart)*SpeedJumpMul)/TimeScalar)*2.0*PI)/TrajPeriod -
PX))/XScaleAbsMax);
Ya = TrajSize*(((A1y*cos(((real_T)StateIndex +
(real_T)PathStart)*SpeedJumpMul)/TimeScalar)*2.0*PI)/TrajPeriod +
A2y*cos(hy(((real_T)StateIndex +
(real_T)PathStart)*SpeedJumpMul)/TimeScalar)*2.0*PI)/TrajPeriod -
PY))/YScaleAbsMax);
}
StateIndex++;

if (StateIndex > 10000) {
StateIndex = 1; // Reset State
}

PropExpState = STATE_EC_TRIAL_DONE;
}

if (StateIndex > iTrialTime + iGoCueTime) { // Limit
StateIndex = 1; // Reset State

PropExpState = STATE_EC_TRIAL_DONE;

RotX = RotMatrix[0][0]*Xa + RotMatrix[0][1]*Ya;
RotY = RotMatrix[1][0]*Xa + RotMatrix[1][1]*Ya;
y[22] = RotX; // XPosTarget
y[23] = RotY; // YPosTarget

// if ((xhs<((Xa+Home_X) + 0.01)) && (xhs>((Xa+Home_X) - 0.01)) && (yhs<((Ya+Home_Y) + 0.01)) && (yhs>((Ya+Home_Y) - 0.01)))) {
% if (HitboxRadius = 0.01; //using a circle function r^2=(X-
(a)^2 + (Y-b)^2 to create a hitbox for the target where r = HitboxRadius.
// if (HitboxRadius = sqrt((((xhs-(RotX+Home_X))^2) + ((yhs-(RotY+Home_Y))^2)))
// StateIndex = 1; // Reset counter
PropExpState = STATE_EC_TRIAL_HOLD; // go to next
}

break;

default: tempfx = 0; tempfy = 0; // output zero torque
break;
}
// from switch (PropExpState)
// SAFETY CHECK START
// DO NOT EDIT!

// Torque Check
if (fabs(y[1]) > MAX_MOTOR_TORQUE || fabs(y[0]) > MAX_MOTOR_TORQUE){
    //tempfx = y[0]; // Send failed torques for debugging or review
    //tempfy = y[1];
    y[0] = y[1] = 0; // Zero Torque
    StateIndex = 0; // Reset State Index
    PropExpState = STATE_SAFETY; // Enter safety state, dead trial
}

// Position Check (only if forces are being applied)
if (fabs(y[1]) != 0 && fabs(y[0]) != 0){
    // Forces are being applied
    if ((xhs-MAX_POSITIONNEG_X) || (xhs-MAX_POSITIONPOS_X) || (yhs-MAX_POSITIONNEG_Y) || (yhs-MAX_POSITIONPOS_Y)){
        //tempfx = y[0]; // Send failed torques for debugging or review
        //tempfy = y[1];
        y[0] = y[1] = 0; // Zero Torque
        StateIndex = 0; // Reset State Index
        PropExpState = STATE_SAFETY; // Enter safety state, dead trial
    }
}

// Send output
y[7] = tempfx;
y[8] = tempfy;
y[6] = count;
y[19] = MaxVy; // Send the maximum velocity for UDP

count++;
}

#define MDL_UPDATE
/* Function: mdlUpdate ========================================================
Abstract: xdot =
*/
static void mdlUpdate(SimStruct *S, int_T tid)
{
    real_T *x = ssGetRealDiscStates(S);
    InputRealPtrsType uPtrs = ssGetInputPortRealsSignalPtrs(S, 0);

    UNUSED_ARG(tid); /* not used in single tasking mode */
}

/* Function: mdlTerminate =====================================================
Abstract: No termination needed, but we are required to have this routine.
*/
static void mdlTerminate(SimStruct *S)
{
    count = 1;
    Enable_flag = 0;
    /*UNUSED_ARG(S); /* unused input argument */
}
#endif MATLAB_MEX_FILE /* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
Appendix C: Sum of Gaussians

% Find_ECDelay_Distances calls FindECDelay function in a loop, calculates
% Error Correction Latencies, Initial Angle, Number of Gaussians etc.
% and stores it for later analysis
clear all
% Parameters to store
NUMBER_OF_TRIALS = 300;
Initial_Angle = zeros(NUMBER_OF_TRIALS,1);
Correction_Angle = zeros(NUMBER_OF_TRIALS,1);
VAF = zeros(NUMBER_OF_TRIALS,1);
N_GAUSS = zeros(NUMBER_OF_TRIALS,1);
Expected_Interception = zeros(NUMBER_OF_TRIALS,1);
MovementOnset = zeros(NUMBER_OF_TRIALS,1);
PeakSpeed = zeros(NUMBER_OF_TRIALS,1);
Gaussians = zeros(3,3,NUMBER_OF_TRIALS); % amplitude, center of Gaussian relative
% to movement onset, SD of the Gaussian
% movement onset is calculated based on n10% of peak speed.
PlannedDistance = zeros(NUMBER_OF_TRIALS,1);
RealizedDistance = zeros(NUMBER_OF_TRIALS,1);
IntededInitialDirection1 = zeros(NUMBER_OF_TRIALS,1);
IntededInitialDirection2 = zeros(NUMBER_OF_TRIALS,1);

for rasi = 11:300
    FileName = ['trial' num2str(rasi) '_Mode50_2020-11-02.mat'];
    try
        [VAF(rasi),N_GAUSS(rasi),MO,TCT,G_parms,
        PeakSpeed(rasi),XPosCorr,YPosCorr,DISTANCE,RDISTANCE,ICD1,ICD2] = FindECDelay
                             (FileName);

        %First Attempt Success
        if N_GAUSS(rasi) == 1
            MovementOnset(rasi) = MO;
            Gaussians(1,1:3,rasi)=G_parms(1:3); % primary movement
            Gaussians(2,1:3,rasi)=[NaN NaN NaN]; % there is no EC
            Correction_Angle(rasi) = NaN;
            %Get initial Angle
            X0 = 0;
            Y0 = 0;
            Directiontime = round(Gaussians(1,2,rasi)) + MO; %Get initial Angle by
            creating vector between home position and peak of 1st Gaussian
            X1 =(-XPosCorr(Directiontime));
            Y1= (YPosCorr(Directiontime));
            origin = atan2(Y0, X0);
            angle1 = -atan2(Y1-Y0, X1-X0);
            Initial_angle_Deg = radtodeg(angle1);
            PlannedDistance(rasi) = DISTANCE;
            RealizedDistance(rasi) = RDISTANCE;
            IntededInitialDirection1(rasi) = ICD1;
            IntededInitialDirection2(rasi) = ICD2;
        end
    end
end
%Correcting for negative angles
if Initial_angle_Deg < 0
    Initial_angle_Deg = Initial_angle_Deg + 360;
end
Initial_Angle(rasi) = Initial_angle_Deg;
% Trials with error corrections
else
    Expected_Interception(rasi) = NaN;
    MovementOnset(rasi) = MO;
    Gaussians(1,1:3,rasi) = G_parms(1:3); % primary movement
    Gaussians(2,1:3,rasi) = G_parms(4:6); % 1st EC
    PlannedDistance(rasi) = DISTANCE;
    RealizedDistance(rasi) = NaN;
    % Get initial Angle
    X0 = 0;
    Y0 = 0;
    Directiontime = round(Gaussians(1,2,rasi)) + MO;
    X1 = (-XPosCorr(Directiontime));
    Y1 = (YPosCorr(Directiontime));
    origin = atan2(Y0, X0);
    angle1 = -atan2(Y1-Y0, X1-X0);
    Initial_angle_Deg = radtodeg(angle1);
    IntendedInitialDirection1(rasi) = ICD1;
    IntendedInitialDirection2(rasi) = ICD2;
    if Initial_angle_Deg < 0
        Initial_angle_Deg = Initial_angle_Deg + 360;
    end
    Initial_Angle(rasi) = Initial_angle_Deg;
end

catch ME;
end
% pause;
end

EndOfFirstGaussian = Gaussians(1,2,:) + 2*Gaussians(1,3,:); % Peak + 2 SD marks the end of 1st Gauss
StartOfFirstG = Gaussians(1,2,:) - 2*Gaussians(1,3,:); % Peak - 2SD is start of Gaussian
StartOfFirstEC = Gaussians(2,2,:) - 2*Gaussians(2,3,:); % 2nd Peak -2 SD is start of EC
NumberOfSuccesses = sum(isnan(StartOfFirstEC))
Delta_T_Gaussians = StartOfFirstEC - endOfFirstGaussian;
Delta_T_Gaussians = squeeze(Delta_T_Gaussians);
Selected_TC_Trials = find(Expected_Interception > 0); % These are the trials where the subject hit the target on the first try without EC
Selected_EC_Trials = find(Delta_T_Gaussians > 0); % These are the ECs with the correction coming after the 1st try ends
Median_Target_Capture_Time = median(Expected_Interception(Selected_TC_Trials))
EC_Delay = StartOfFirstEC - StartOfFirstG; % Calculates the EC latency of each trial
EC_Delay = squeeze(EC_Delay);
Peak_Peak_Distance = Gaussians(2,2,:); %Distance peak Gaussian Peaks
Peak_Peak_Distance = squeeze(Peak_Peak_Distance);

%Find Amplitude of peaks
Gauss_1Height = Gaussians(1,1,:);
Gauss_2Height = Gaussians(2,1,:);
Gauss_1Height = squeeze(Gauss_1Height);
Gauss_2Height = squeeze(Gauss_2Height);

% Function FindECDelay processes single trial data (Speed, Power, Time, % Distance etc..)
function [VAF,N_GAUSS,MO,TCT,rasp,PeakSpeed,XPosCorr,YPosCorr,DISTANCE,RDISTANCE, ICD1, ICD2] = FindECDelay (FileName)

load(FileName);
[b,a]=butter(2,10/500);
[d,c]=butter(4,0.05/500,'high');

%Basic Data
TargState = (Trial.TargState);
TrialLoad = find(TargState==22);
LoadTime = TrialLoad(end); %trial loading time
TrialStart = find(TargState==2); %when does trial begin
TrialEnd = find(TargState==27); %determine when the trial ends
TargCap = find(TargState==26);
if TargCap>=0;
   TargCap = find(TargState==26); %When Targ is captured
else
   TargCap = TrialEnd;
end

TargCapDot = TargCap(1,1)-LoadTime; %correct for state 22 (loading time),
GoCue = (Trial.GoCue);
GoCueStart = find(GoCue==1); %determine when the go cue occurs
GoCueTime = GoCueStart(1,1)-LoadTime; %gives time at which go cue happens

TargX = (Trial.XPosVis)*100; TargX = TargX(TrialStart(1,1):TrialEnd(1,1)); %XPosition in cm
TargY = (Trial.YPosVis)*100; TargY = TargY(TrialStart(1,1):TrialEnd(1,1)); %YPosition in cm
XPos = (Trial.XPos)*100; XPos = XPos(TrialStart(1,1):TrialEnd(1,1)); %XPosition in cm
YPos = (Trial.YPos)*100; YPos = YPos(TrialStart(1,1):TrialEnd(1,1)); %YPosition in cm
Time = (Trial.Time);
% Calculate Hand Power
```matlab
I = find(Trial.XPos); I = I(1:end-20);
HandFx = (Trial.RLCHandFx(I)); HandFx = HandFx - mean(HandFx(1:50));
HandFy = -(Trial.RLCHandFy(I)); HandFy = HandFy - mean(HandFy(1:50));
HandFx = filtfilt(b,a,HandFx);
HandFy = filtfilt(b,a,HandFy);
HandForce = HandFx + i*HandFy;
HandForceVector = [HandFx HandFy]';
HX = Trial.XPos(I);
HY = -Trial.YPos(I);

VX = diff(HX)*1000;
VY = diff(HY)*1000;

HandV = [VX VY];
Power = HandV(:,:,1) * HandForceVector(:,1:end-1);
MyPower = diag(Power); %in Nm/s

% Determine Movement Onset: (MO)
GoCueSignal = GoCue(TrialStart(1,1):TrialEnd(1,1));
GoCueSignal = GoCueSignal*30;
XPosCorr = (XPos - 14); %Correcting for home position in cm;
YPosCorr = (YPos - 59.25); %Correcting for home position in cm
Xinc = diff(XPosCorr);
Yinc = diff(YPosCorr);
Distance = sqrt(((Xinc).^2)+((Yinc).^2));
Speed = Distance/.001; %in cm/s
PeakSpeed = max(Speed);
PeakSpeedTime = find(Speed==PeakSpeed); %time at which max speed occurs
ThresholdSpeed = PeakSpeed*.11; %setting speed for a reaction threshold
ThresholdLow = PeakSpeed*.1;
PrePeak = Speed(1:PeakSpeedTime(1,1)); %getting the speeds before max speed
PreRT = find((ThresholdLow < PrePeak) & (PrePeak < ThresholdSpeed)); %all times where the speed is lower than the threshold
RT = PreRT(1,1); % time at which speed crosses the threshold
TrueRT = RT - GoCueTime-1000; % The Go signal happens 1000 ms after GoCueTime, aligned with 4th beep.
MO = RT;

% Determine time of target capture: TCT
TCT = TargCapDot;

%find movement extent
TCXPosition = TargX(GoCueTime+1410); %Target x position at 410ms after Go
TCYPosition = TargY(GoCueTime+1410); %Target y position at 410ms after Go
DVector = [TCXPosition, TCYPosition;0,0];
DISTANCE = pdist(DVector,'euclidean');

TCXCapture = TargX(TargCapDot);
TCYCapture = TargY(TargCapDot);
```
RealizedDVector = [TCXCapture, TCYCapture;0,0];
RDISTANCE = pdist(RealizedDVector,'euclidean');

%Find Angle between home position and target position
XT0 = 0;
YT0 = 0;
XT1 = (-TargX(GoCueTime+150)); % X target at 150ms after go cue per Mrotek 2013
YT1 = (TargY(GoCueTime+1150)); % Y target at 150ms after go cue per Mrotek 2013

angle = -atan2(YT1-YT0, XT1-XT0);
ICD1 = radtodeg(angle);
if ICD1<0
    ICD1 = ICD1 + 360;
end

XT2 = (-TargX(GoCueTime+1410)); % X target at 410ms after go cue per TCT
YT2 = (TargY(GoCueTime+1410)); % X target at 410ms after go cue per TCT
angle2 = -atan2(YT2-YT0, XT2-XT0);
ICD2 = radtodeg(angle2);
if ICD2<0
    ICD2 = ICD2 + 360;
end

% determine Number of Gaussians to fit
if (0)
    figure(1)
    hold on
    plot(MO:TCT,MyPower(MO:TCT),'k')
    plot(MO,MyPower(MO),'g.','MarkerSize',12)
    plot(TCT,MyPower(TCT),'r.','MarkerSize',12)
    hold off
end

MO_TCT_Power = MyPower(MO:TCT);

%Filter Power
dPowerDT = filtfilt(b,a,diff(MyPower(MO:TCT))*1000);
ddPowerDT = filtfilt(b,a,diff(dPowerDT)*1000);
Sign_ddPowerDT = sign(ddPowerDT);
dSign = diff(Sign_ddPowerDT);
InPoint = find(abs(dSign));
Total_zero_crossing = length(InPoint);

if Total_zero_crossing == 0 % if there are no zero crossings detected, assume 1 submovement and populate values to develop IC
    Total_zero_crossing = 1;
    [M,I] = max(velocity);
end
N_GAUSS = round(Total_zero_crossing/2);

%Find Initial Conditions of N gaussian
Gauss_index = zeros(N_GAUSS,1);
for pabi = 1:N_GAUSS
    odd_array = (pabi-1)*2+1;
    Gauss_index(pabi) = InPoint(odd_array);
end

IC = 0*ones(1,3*N_GAUSS);
for pabj = 1:N_GAUSS
    IC(3*pabj-2) = MO_TCT_Power(Gauss_index(pabj)); % amplitude, based on v_peak during jerk trough
    IC(3*pabj-1) = Gauss_index(pabj); % center of Gaussian, based on location of v_peak during jerk trough
    IC(3*pabj) = 50; % sigma - assume standard deviation is 1/4 of space between peaks
end

%%%%%%

% Fit N gaussians to Power between MO and TCT
[rasp,Model_sum,VAF] = Aim2_FitSubmovementsGaussianSum_ver5(MyPower,MO,TCT,N_GAUSS,IC);

% Plot the data and gaussian fit and display VAF in command winndow
disp (['File ' FileName '; VAF: ' num2str(VAF) ', N of Gauss: ' num2str(N_GAUSS)]);

function output = Aim2_nGaussSum(p,Samples,vel,n_gauss)
% output = Aim2_nGaussSum(p,Samples,vel,n_gauss)
% Aim2_nGaussSum calculates mean squared error between a vector (vel) of length (Samples) and the sum of n Gaussians
% p = parameters (1x3*n_gauss vector) of the form: (amplitude_1, mean_1, standard deviation_1, ... , amplitude_n_gauss, mean_n_gauss, standard deviation_n_gauss)
% Samples = length(vel) vector for reconstructing Gaussians
% vel = data to be modeled
% n_gauss = number of Gaussians to be fit

% initialize arrays:
amplitude   = zeros(n_gauss,1);
mu          = zeros(n_gauss,1);
sigma = zeros(n_gauss,1);  
gauss = zeros(n_gauss,length(Samples));  

% create Gaussian of time series:  
for mcbk = 1:n_gauss  
    amplitude(mcbk,1) = p(3*mcbk-2);  
    mu(mcbk,1) = p(3*mcbk-1);  
    sigma(mcbk,1) = p(3*mcbk);  
    gauss(mcbk,:) = amplitude(mcbk)*exp(-(Samples-mu(mcbk)).^2/(2*sigma(mcbk)^2));  
end  

% calculate sum of Gaussians:  
total_sum = sum(gauss,1);  

% calculate squared error of data - model for output:  
output = sum((vel-total_sum).^2);  

if(0)  
    figure  
    hold on  
    plot(vel,'k')  
    plot(gauss,'r')  
end

function output = Aim2_nGaussSum(p,Samples,vel,n_gauss)  

% output = Aim2_nGaussSum(p,Samples,vel,n_gauss)  
%  
% Aim2_nGaussSum calculates mean squared error between a vector (vel)  
% of length (Samples) and the sum of n Gaussians  
%  
% p = parameters (1x3*n_gauss vector) of the form: (amplitude_1, mean_1,  
% standard deviation_1, ..., amplitude_n_gauss, mean_n_gauss,  
% standard deviation_n_gauss)  
%  
% Samples = length(vel) vector for reconstructing Gaussians  
%  
% vel = data to be modeled  
%  
% n_gauss = number of Gaussians to be fit  

% initialize arrays:  
amplitude = zeros(n_gauss,1);  
mu = zeros(n_gauss,1);  
sigma = zeros(n_gauss,1);  
gauss = zeros(n_gauss,length(Samples));
% create Gaussian of time series:
for mcbk = 1:n_gauss
    amplitude(mcbk,1) = p(3*mcbk-2);
    mu(mcbk,1) = p(3*mcbk-1);
    sigma(mcbk,1) = p(3*mcbk);
    gauss(mcbk,:) = amplitude(mcbk)*exp(-(Samples-
        mu(mcbk)).^2/(2*sigma(mcbk)^2));
end

% calculate sum of Gaussians:
total_sum = sum(gauss,1);

% calculate squared error of data - model for output:
output = sum((vel-total_sum).^2);

if(0)
    figure
    hold on
    plot(vel,'k')
    plot(gauss,'r')
end

function [N_GAUSS, IC] = Aim2_Find_NGauss_IC(MOVE_START,MOVE_STOP,JERK,VELOCITY)
% [nGauss,IC] = Aim2_Find_NGauss_IC(MOVE_START,MOVE_STOP,JERK)
% This function searches for likely submovements within the movement period
% defined in Aim2_FiducialPoints_Kinematic_ver11.m using the number of
% zero crossings in the Jerk time series to identify probable submovements
% in the Velocity trace. Generally, a submovement will have 2
% zero-crossings in the Jerk trace, though this number may be slightly
% lower when there are a few submovements (hence, the value is rounded up if
% there is an odd number of zero crossings).

% Define search window:
jerk = JERK(MOVE_START:MOVE_STOP);
velocity = VELOCITY(MOVE_START:MOVE_STOP);

% Determine number of zero crossings in jerk signal:
jerk_sign = sign(jerk);
jerk_sign_diff = diff(jerk_sign);
jerk_pos_zero_cross = find(jerk_sign_diff > 1);
n_pos_zero_cross = length(jerk_pos_zero_cross);
jerk_neg_zero_cross = find(jerk_sign_diff < -1);
n_neg_zero_cross = length(jerk_neg_zero_cross);
total_zero_cross = n_pos_zero_cross + n_neg_zero_cross;

% Determine estimated number of submovements (number of Gaussians to use in
% fit):
if total_zero_cross == 0 % if there are no zero crossings detected, assume 1 submovement and populate values to develop IC
    total_zero_cross = 1;
    [M,I] = max(velocity);
end
N_GAUSS = round(total_zero_cross/2);

% Determine locations of velocity peaks during jerk troughs to use as IC
% to better likelihood of finding global (as opposed to local) minimum.
temp_ind = find(jerk_sign_diff ~= 0);
indices = [1; temp_ind; length(jerk_sign)]';

% create trough index matrix
mcbj = 1;
for mcbk = 1:length(indices)-1
    if sum(jerk_sign(indices(mcbk):indices(mcbk+1))) < 0
        [M(mcbj),I_temp] = max(velocity(indices(mcbk):indices(mcbk+1)));
        I(mcbj) = I_temp + indices(mcbk);
        start_ind(mcbj) = indices(mcbk);
        stop_ind(mcbj) = indices(mcbk+1);
        mcbj = mcbj + 1;
    else
    end
end

% initialize IC matrix:
IC = 0*ones(1,3*N_GAUSS);

% %%%%%%%%% Create initial conditions matrix: %%%%%%%%%
% IC = [amplitude_1 mu_1 sigma_1 ... amplitude_n mu_n sigma_n];
for mcbi = 1:N_GAUSS
    IC(3*mcbi-2) = M(mcbi); % amplitude, based on v_peak during jerk trough
    IC(3*mcbi-1) = I(mcbi); % center of Gaussian, based on location of v_peak during jerk trough
    IC(3*mcbi) = 0.5*(stop_ind(mcbi)-start_ind(mcbi)); % sigma - assume standard deviation is 1/4 of space between peaks
end

function [P,Model_sum,VAF] = Aim2_FitSubmovementsGaussianSum_ver5(VELOCITY,MOVE_START,MOVE_STOP,N_GAUSS,IC)

% Aim2_FitSubmovementsGaussianSum_ver5.m
% % [P] = Aim2_FitSubmovementsGaussianSum_ver5(VELOCITY,MOVE_START,MOVE_STOP,N_GAUSS)
% % P is the set of parameters that are being optimized. In this case, that
% % includes an amplitude, mean, and standard deviation value for each
% % Gaussian.
%
% VELOCITY is the time-series of the velocity of a movement. This vector
% must be positive for proper fit of Gaussians.
% MOVE_START is the time of movement start.
% MOVE_STOP is the time of movement stop.
% N_GAUSS is the number of Gaussians to fit to the data
% IC is the initial conditions matrix.
%
% This function runs a constrained, non-linear optimization with the
% objective of minimizing mean squared error between the velocity
% trace and the sum of n Gaussians.

%%%%%% define range %%%%%%%
RANGE_VELOCITY = MOVE_START:MOVE_STOP;
RANGE = 1:length(RANGE_VELOCITY);
VELOCITY = VELOCITY(RANGE_VELOCITY)';  % velocity in degrees/second

%%%%%% initialize variables %%%%%%%
options = optimset('MaxFunEvals',7500); % set number of function evaluations
from 3000 (default) to 7500

%%%%%% Setup Constraints %%%%%%%
for mcbi = 1:N_GAUSS
    lb(1,3*mcbi-2:3*mcbi) = [0 0 50];  % lower bounds for [amplitude, mu (mean),
    sigma (standard deviation)]
    ub(1,3*mcbi-2:3*mcbi) = [max(VELOCITY), length(RANGE),
    (1/N_GAUSS)*(length(RANGE)/2)];  % relaxed constraint on standard deviation
end

%%%%%% Run non-linear optimization with n_gauss gaussians: %%%%%%%%
P = fmincon(@(P)
    Aim2_nGaussSum(P,RANGE,VELOCITY,N_GAUSS),IC,[],[],[],[],lb,ub,[],options);

%%%%%% initialize arrays:
amplitude = zeros(N_GAUSS,1);
mu = zeros(N_GAUSS,1);
sigma = zeros(N_GAUSS,1);
gauss = zeros(N_GAUSS,length(RANGE));

%Model = P(1)*exp(-(RANGE-P(2)).^2/(2*P(3)^2));
for j = 1:N_GAUSS
    amplitude(j,1) = P(3*j-2);
    mu(j,1) = P(3*j-1);
    sigma(j,1) = P(3*j);
    gauss(j,:) = amplitude(j)*exp(-(RANGE-mu(j)).^2/(2*sigma(j)^2));
end

% calculate sum of Gaussians:
Model_sum = sum(gauss,1);

if(0)
    figure(1)
xlabel('Time (ms)'), ylabel('Power (Nm/s)')
hold on
plot(VELOCITY,'k')
for i = 1:N_GAUSS
    Model = p(3*i-2)*exp(-(RANGE-P(3*i-1)).^2/(2*p(3*i)^2));
    plot(Model,'r')
hold on
end
plot(Model_sum,'g')
end
Resid = VELOCITY - Model_sum;
VarData = var(VELOCITY);
VarResid = var(Resid);
VAF = 1-(VarResid/VarData);

function [VAF,N_GAUSS_revised,P_revised,MANUAL_ADJUST] = Aim2_CheckGaussFit(P,N_GAUSS,POSITION,VELOCITY,JERK,MOVE_START,MOVE_STOP,POS_START,TYPE)

%%%%%%%%% Define relevant time series: %%%%%%%%%

%%%%%%%%% define range %%%%%%%
RANGE_VELOCITY = MOVE_START:MOVE_STOP;
RANGE = 1:length(RANGE_VELOCITY);
POSITION = POSITION(RANGE_VELOCITY);% position in degrees (time interval is ms)
VELOCITY = 1*VELOCITY(RANGE_VELOCITY)';% velocity in degrees/second
JERK = JERK(RANGE_VELOCITY);

%%%%%%%%% set constants %%%%%%%%%
SCRSZ = get(0,'ScreenSize');
JERK_COEFF = 1/1000;
POS_COEFF = 0.001;
jerk_y_shift = -1.1*max(JERK_COEFF*JERK);
pos_y_shift = 1.1*max(VELOCITY);

%%%%%%%%% Initialize arrays %%%%%%%%%
AMPLITUDE = 0*ones(1,N_GAUSS);
MEAN = 0*ones(1,N_GAUSS);
STDEV = 0*ones(1,N_GAUSS);
GAUSS = 0*ones(N_GAUSS,length(RANGE));

%%%%%%%%% Reconstruct sum of Gaussians %%%%%%%%%
% Separate out parameters returned from fmincon function:
% amplitude:
for mcbi = 1:N_GAUSS
    AMPLITUDE(mcbi) = P(3*mcbi-2);
    MEAN(mcbi) = round(P(3*mcbi-1));
    STDEV(mcbi) = P(3*mcbi);
end
% calculate Gaussian curves
for mcbi = 1:N_GAUSS
    gauss_temp = AMPLITUDE(mcbi)*exp(-(RANGE-MEAN(mcbi)).^2/(2*STDEV(mcbi)^2));
    GAUSS(mcbi,:) = gauss_temp;
end

% calculate sum of Gaussian curves
sum_of_gaussians = sum(GAUSS,1);

% calculate variance accounted for (VAF)
var_data    = var(VELOCITY);
var_error   = var(sum_of_gaussians-VELOCITY);
VAF         = 100*(1-(var_error/var_data));

% Plot fits, sum and data:
figure('Position',[(SCRSZ(3)/3)+1 SCRSZ(4) SCRSZ(3)/2 SCRSZ(4)]);
hold on
for mcbi = 1:N_GAUSS
    plot(GAUSS(mcbi,:))
end
plot(VELOCITY,'k','LineWidth',2)
plot(sum_of_gaussians)

% plot jerk to make sure correct number of Gaussians were used for fit:
plot(JERK_COEFF*JERK+jerk_y_shift)

% plot position and cumulative sum of gaussians to view model's applicability to position data:
plot(POSITION+pos_y_shift,'k','LineWidth',2)
cumsum_gausssum = POS_START+POS_COEFF*0*ones(1,length(RANGE_VELOCITY));
title(strcat('N Gauss = ',num2str(N_GAUSS),' ; VAF = ',num2str(VAF),'...LEFT CLICK TO ACCEPT; RIGHT CLICK TO MANUALLY DEFINE N AND IC'))

% Verify that optimization performed as expected:
% RIGHT CLICK (1): ACCEPT MODEL
% LEFT CLICK (3) : ENTER N, CLICK N TIMES TO DETERMINE LOCATIONS/AMPLITUDES FOR
% PEAK IC
[-,-,ACCEPT_MODEL] = ginput(1);

% IF MODEL IS ACCEPTED, RETURN ZERO:
N_GAUSS_revised = 0;
VAF_revised = 0;
P_revised = 0;
MANUAL_ADJUST = 0;
while ACCEPT_MODEL ~= 1
    MANUAL_ADJUST = 1;
    prompt = 'How many Gaussians should be fit to these data?';
    N_GAUSS_revised = input(prompt);
    title(strcat('CLICK ',num2str(N_GAUSS_revised),' TIMES AT APROXIMATE VELOCITY PEAKS TO DETERMINE IC VALUES FOR MODEL FIT:'))
    [I,M] = ginput(N_GAUSS_revised); % I is index value (x), M is magnitude (y) of cursor clicks
    indices = [1,I'];

    % %%%%%%%%%% Create initial conditions matrix: %%%%%%%%%%
    % initialize IC matrix:
    IC = 0*ones(1,3*N_GAUSS_revised);
    % IC = [amplitude_1 mu_1 sigma_1 ... amplitude_n mu_n sigma_n];
    for mcbi = 1:N_GAUSS_revised
        IC(3*mcbi-2) = M(mcbi); % amplitude, based on v_peak during jerk trough
        IC(3*mcbi-1) = I(mcbi); % center of Gaussian, based on location of v_peak during jerk trough
        IC(3*mcbi) = 0.5*(indices(mcbi+1)-indices(mcbi)); % sigma - assume standard deviation is 1/4 of space between peaks
    end

    % %%%%%%%%%% RE-RUN CONSTRAINED NONLINEAR OPTIMIZATION: %%%%%%%%%%%%%
    % initialize variables
    options = optimset('MaxFunEvals',7500); % set number of function evaluations from 3000 (default) to 7500
    % Setup Constraints
    for mcbi = 1:N_GAUSS_revised
        lb(1,3*mcbi-2:3*mcbi) = [0 0 50]; % lower bounds for [amplitude, mu (mean), sigma (standard deviation)]
        ub(1,3*mcbi-2:3*mcbi) = [max(VELOCITY), length(RANGE), (1/N_GAUSS_revised)*(length(RANGE)/2)]; % relaxed constraint on standard deviation
    end
    % Run non-linear optimization with N_GAUSS_revised gaussians:
    P_revised = fmincon(@(P_revised)Aim2_nGaussSum(P_revised,RANGE,VELOCITY,N_GAUSS_revised),IC,[],[],[],[],lb,ub,[],options);

    % %%%%%%%%%% Reconstruct sum of Gaussians %%%%%%%%%%
    % Separate out parameters returned from fmincon function:
    % amplitude:
    % %%%%%%%%%% Initialize arrays %%%%%%%%%%
    AMPLITUDE_revised = 0*ones(1,N_GAUSS_revised);
    MEAN_revised = 0*ones(1,N_GAUSS_revised);
    STDEV_revised = 0*ones(1,N_GAUSS_revised);
    GAUSS_revised = 0*ones(N_GAUSS_revised,length(RANGE));
    for mcbi = 1:N_GAUSS_revised
        AMPLITUDE_revised(mcbi) = P_revised(3*mcbi-2);
        MEAN_revised(mcbi) = round(P_revised(3*mcbi-1));
        STDEV_revised(mcbi) = P_revised(3*mcbi);
for mcbi = 1:N_GAUSS_revised
    gauss_temp = AMPLITUDE_revised(mcbi)*exp(-(RANGE_MEAN_revised(mcbi)).^2/(2^STDEV_revised(mcbi)^2));
    GAUSS_revised(mcbi,:) = gauss_temp;
end

sum_of_gaussians_revised = sum(GAUSS_revised,1);

var_data = var(VELOCITY);
var_error = var(sum_of_gaussians_revised-VELOCITY);
VAF_revised = 100*(1-(var_error/var_data));

% Plot fits, sum and data:
figure('Position',[(SCRSZ(3)/3)+1 SCRSZ(4) SCRSZ(3)/2 SCRSZ(4)])
hold on

% plot velocity, sum of Gaussians, and individual Gaussians to view model
% fit:
for mcbi = 1:N_GAUSS_revised
    plot(GAUSS_revised(mcbi,:))
end
plot(VELOCITY,'k','LineWidth',2)
plot(sum_of_gaussians_revised)

cumsum_gauss_sum_revised = POS_START+POS_COEFF*0*ones(1,length(RANGE_VELOCITY));

title(strcat('N Gauss = ',num2str(N_GAUSS_revised),'; VAF = ','num2str(VAF_revised),',...LEFT CLICK TO ACCEPT; RIGHT CLICK TO MANUALLY DEFINE N AND IC'))

% Verify that optimization performed as expected:
% RIGHT CLICK (1): ACCEPT MODEL
% LEFT CLICK (3): ENTER N, CLICK N TIMES TO DETERMINE LOCATIONS/AMPLITUDES
% FOR PEAK IC
[~,~,ACCEPT_MODEL] = ginput(1);