JOHNS HOPKINS UNIVERSITY

Optimizing Dynamic Control Systems: Simulink Workstreams, Automated Code Generation, and Latency Analysis in Robotics Middleware

TABLE OF CONTENTS

ABSTRACT

This paper outlines a comprehensive approach to leveraging Simulink for developing dynamic control systems, exemplified by surgical robotics. By employing Simulink's robust modeling, simulation, and analysis capabilities, we present a streamlined workstream that facilitates precise control system design and enhances the development process for complex robotic systems. Additionally, the paper introduces an automated code generation workflow designed to minimize repetitive coding tasks, enabling developers to focus on product-specific code. This process employs Python's Jinja2 template engine to dynamically generate MATLAB (.m) and target-specific C++ (.tlc) files, ensuring consistency and efficiency across large-scale communication infrastructures.
Furthermore, the paper investigates the latency performance of ROS 2 and Connext in two transmission scenarios: Shared Memory Inter-Process Communication (SHMIO) and IPv4 UDP. Experiments reveal that while ROS 2 excels in abstraction and versatility, Connext consistently demonstrates superior efficiency with lower latency across various payload sizes and transmission frequencies. These findings highlight critical trade-offs in middleware selection for latency-sensitive applications, such as real-time robotics. Together, these contributions establish a foundation for efficient, scalable, and performance-driven development workflows in dynamic control systems.

INTRODUCTION

Simulink is a critical tool for engineers, offering a robust environment for modeling, simulating, and analyzing dynamic systems with precision and efficiency. It is particularly important in industries where testing physical prototypes is costly or risky. Studies indicate that model-based design, a hallmark of Simulink, can reduce development time by up to 50% and lower costs by 30%, highlighting its value in streamlining workflows.[1] In robotics, Simulink plays a pivotal role by enabling the design and simulation of control algorithms, kinematics, and dynamics of robotic systems. Engineers can model robotic arms, autonomous vehicles, or drones, simulating their behavior in virtual environments to optimize performance before hardware implementation. Its integration with real-time hardware like Arduino, Raspberry Pi, and ROS (Robot Operating System) further supports prototyping and deployment, making it indispensable in advancing robotic innovation.

Based on the advantages above, Medtronic selects Simulink for developing product software due to its exceptional capability to streamline and enhance the development process for complex robotic systems. Simulink simplifies the derivation of dynamic models for robotic subsystems, such as controlling a motor by sending position or velocity commands at specific frequencies, ensuring precise and efficient performance. Its modular design and graphical interface provide scalability and adaptability, enabling engineers to manage model changes seamlessly and reuse

controllers across different motors by simply adjusting parameters. Additionally, Simulink supports advanced testing through Model-in-the-Loop (MiL) and Hardware-in-the-Loop (HiL) simulations, allowing Medtronic to verify complex mathematical formulas and evaluate algorithms in integrated hardware environments. The integrated code generation feature further solidifies its utility by automatically producing high-quality C/C++ code using tools like Codegen and .tlc templates, ensuring smooth integration with other hardware systems. These capabilities make Simulink an ideal choice for Medtronic to innovate and maintain precision in their robotic solutions.

Another topic will be discussed hereby is the performance of communication. The latency will be compared between a popular robotic framework, ROS2 and one of its communication backbones, Data Distribution Service (DDS) (Connext was used in the experiment). While the original ROS gained immense popularity in academia due to its extensive tools and pre-built packages, its middleware limitations prevented it from being widely adopted in real-time, multi-robot, or critical systems. ROS2 addresses these issues by building on modern standards and frameworks, making it suitable for a broader range of applications, including autonomous vehicles and industrial automation. This evolution has allowed ROS to transition from an academic research tool to a robust, versatile platform for real-world robotics solutions. The integration of the DDS framework into ROS 2 is a key factor behind its enhanced capabilities. DDS, an open-standard communication software framework, provides low latency, high reliability, and extensive Quality of Service (QoS) controls, making it ideal for challenging environments like noisy industrial settings or systems with intermittent connectivity.[2] In this article, we will examine the latency performance of ROS 2 and its underlying DDS framework. As a layered architecture, ROS 2 routes all data communication through its middleware before it reaches the DDS backend, which can introduce considerable latency. While this design ensures flexibility and access to ROS 2's extensive tools, the added layers can create performance bottlenecks, particularly in latency-sensitive applications like real-time robotics and autonomous systems. To mitigate these issues, critical application components can be designed to directly interact with the DDS API, bypassing the ROS 2 middleware. This approach significantly reduces latency while preserving full interoperability with the ROS 2 ecosystem. A compelling example comes from the 2021 Indy Autonomous Challenge, where engineers optimized a ROS 2 LiDAR device driver to communicate directly with DDS.

In the following section, we will introduce an experiment designed to test and compare the latency performance between ROS 2 and DDS. The experiment aims to provide a quantitative analysis of the latency differences caused by the layered architecture of ROS 2, which routes data through its middleware before reaching the DDS framework. By measuring and analyzing these differences, we can better understand the performance trade-offs associated with using ROS 2 and explore how directly leveraging DDS can impact critical application performance.

TERMINOLOGY

XXX_thetas (XXX stands for the name of angle array)	An array with size of 6. Depict 6 joint angles of UR5 goal_thetas: the selected joint angles from the inverse kinematic solutions. cur_thetas: the current joint angles on the motion profile real_thetas: the actual joint angles of the motors in real time. These values are used for comparing with goal_thetas and simultaneously, they are provided to the solution validation module and displayed on the GUI via SHMIO.
Cur_velocity	An array with size of 6. Depict the current joint velocities of the six UR5 joints on the motion profile. These velocities are converted to their corresponding voltages (cur_voltage), as the transfer function requires an applied voltage input.
XXX_voltage (XXX stands for the name of voltage array)	An array with size of 6. Depict 6 voltages applied to each of the UR5 motors cur_voltage: converted from cur_velocity feedback_voltage: converted from the joint velocity output from the transfer function cmd_voltage: the applied voltage to control the joint velocity. It is in sum of the PID controller output and the position feedforward voltage
Position (p)	An array with size of 3. Depict x, y, z position in Cartesian frame.
Orientation (q)	An array with size of 3. Depict the Euler angles in the order of Rx, Ry, Rz
Pose	The Combination of Position and Orientation.
RTI DDS / Connext	Two of them are interchangeable here referring to the RTI's specific implementation of the DDS standard
Topic	A unique identifier used in DDS communication that allows Publishers and Subscribers to connect to the same data stream
Jinja template	a text-based template engine used in Python applications to dynamically generate content by combining static templates with data

DESIGNED SURGICAL ROBOTIC SYSTEM

The Medtronic Surgical Robotic Assisted System (Hugo^{TM} RAS) is a versatile platform designed to translate surgeon inputs into precise robotic movements for advanced surgical procedures. Its architecture centers around three key subsystems: the Surgeon Console (SC), the Tower, and the Arm Carts Assembly (ACA). The SC serves as the interface for the surgeon, featuring controls like handles, foot pedals, and buttons to capture inputs, which are transmitted via DDS for high-level communication. The Tower acts as the central processing hub, housing the Master-Slave Control (MSC) computer that processes commands from the SC and distributes them to the ACA. The ACA comprises multiple Arm Carts, each equipped with a Slave Robot Assembly (SRA) computer that interfaces with controllers for Instrument Drive Units (IDU), Robot Arms (RA), and Setup Arms (SA). These components utilize EtherCAT for real-time joint communication and SHMIO for efficient data exchange within the ACA. This tightly integrated communication architecture ensures seamless coordination across subsystems, enabling precise and reliable robotic operations during surgery.

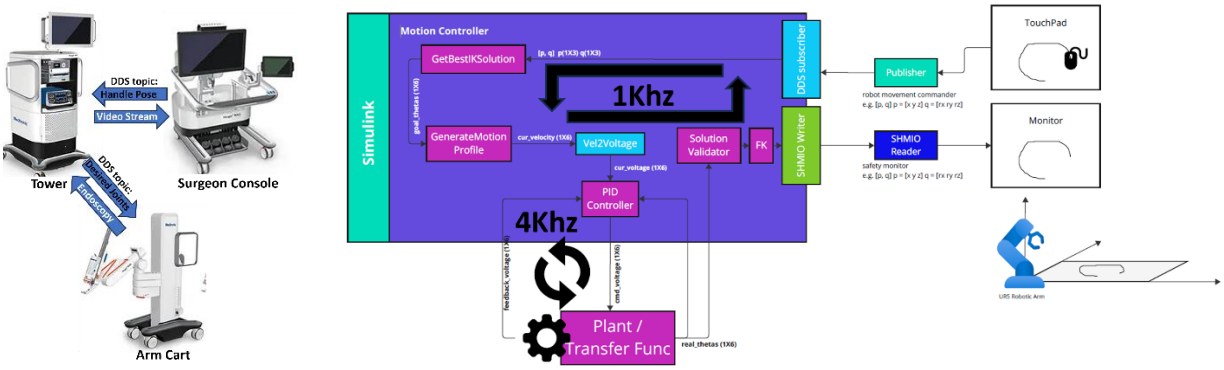


Figure 1 DDS communication architecture of Medtronic Hugo RAS system and the designed surgical robotic system in analogy to Hugo RAS system

Learning from the Hugo RAS system, my system adopts a streamlined architecture for precision control, integrating input capture and real-time execution in a cohesive workflow as shown in *Figure 1*. A touchpad serves as the User Input Device, capturing 2DOF (x, y) cursor positions and combining them with preconfigured constant values for the remaining 4DOF (z, Rx, Ry, Rz) to define the desired pose. A Publisher Node transmits this pose array at 1 kHz to a Simulink-based controller, which computes optimal joint solutions (goal_thetas) through inverse kinematics. The solutions are fed to the Motion Profile Generator, which calculates the current velocity and the corresponding applied voltage in the motion profile. These are subsequently applied to the PID controller operating at 4 kHz, generating precise motor commands (cmd_voltage) for the DC motors driving the robotic arm. The feedback loop ensures real-time monitoring and visualization, with the arm's actual pose transmitted via SHMIO to the GUI (Graphical User Interface). More detailed designs of GUI can be obtained from the Appendix. This tightly integrated system ensures accurate and responsive robotic arm movements, enabling precise execution of tasks.

PROJECT COMPILATION AND AUTO CODE GENERATION

In the Medtronic Hugo™ RAS software application, more than 200 DDS topics are defined, highlighting the sheer scale and complexity of the communication infrastructure in such a sophisticated robotics system. Auto code generation for DDS publishers and subscribers, as well as SHMIO publishers and subscribers, becomes essential in this context. Managing this vast network of topics—across diverse subsystems like UPS units, switches, computing nodes, and gateways—requires a solution that handles the repetitive nature of communication code efficiently. While the main logic of these publishers and subscribers is similar, variations in topics, QoS settings, and custom input/output parameters introduce tedious and error-prone manual

coding challenges. Automating this process ensures consistency, reduces development time, minimizes human error, and enables scalable management of complex communication layers, making it an indispensable approach in the development of advanced surgical robotics applications like Hugo™ RAS.

The auto code generation workflow is depicted shown as below. The process begins by parsing the input definitions into a data structure that encapsulates the required information about block properties, ports, and configurations. Using Python's Jinja2 template engine, predefined templates for .tlc and .m files are populated with this parsed data, ensuring consistency and maintainability. The .m file, a Level-2 MATLAB S-function, is generated to define the graphical properties and behavior of the Simulink block, including input/output ports, parameters, and setup methods such as PostPropagationSetup and WriteRTW. Meanwhile, the .tlc file is generated to define the code translation logic for C++ generation during the Simulink model build.

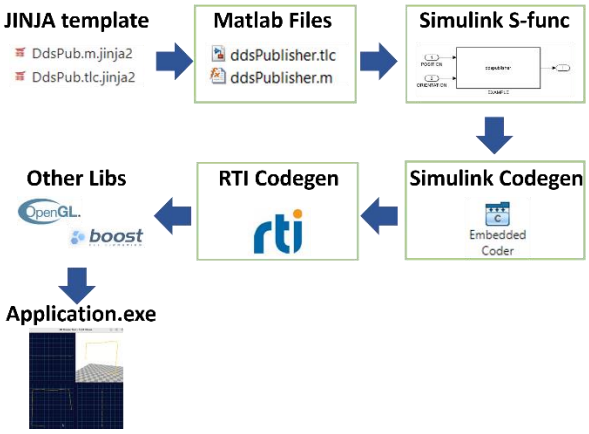


Figure 2 Auto code generation from template to application

Specifically, the process within the green squares highlights the automated code generation workflow in *Figure 2*. Jinja template is a text-based template engine used in Python applications to dynamically generate content by combining static templates with data. In this scenario, .m and .tlc are automatically generated by Jinja templates. .m file defines the Simulink block's behavior, interface, and configuration. .tlc controls how the block is translated into target-specific C++ code during code generation e.g. include necessary Connext library, create DDS participant, topic, write/reader. Subsequently, the .m file associated with a Level-2 S-Function block defines the setup and functionality of the block in MATLAB code. .tlc customizes the code generation process, ensuring the generated C++ code adheres to Connext DDS applications. After, RTI Codegen automatically generates type-specific code from user-defined data structures in IDL (Interface Definition Language). It creates C++ source files that include serialization and deserialization logic, as well as APIs for publishing, subscribing, and managing DDS entities like topics and data writers.

SIMULINK MODELS

In *Figure 3*, the Simulink model for the robot motion control demonstrates a clear data flow architecture, starting with the ddsPoseSub block, which receives the goal_pose from the touchpad system. This data, along with original_thetas, is processed in the MotionController block to compute the real_thetas, which can be joint angles from UR5 joint position sensors. These angles are then passed through the ForwardKinematics block to calculate a 4x4 homogeneous transformation matrix representing the spatial position and orientation. The SolutionValidator block validates this matrix, extracting the real_position and real_orientation to ensure accuracy and compliance with constraints. Finally, these validated outputs are published to the GUI via the shmioPosePub block, providing users with real-time feedback on the robot's actual motion state.

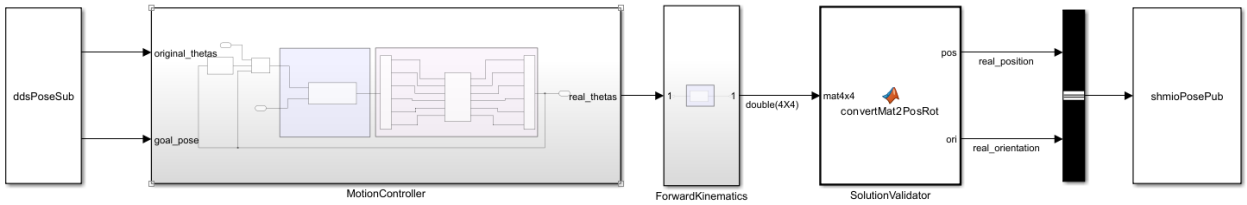


Figure 3 Top layer of Simulink model

The MotionController consists of two main components that will be detailed in the following sections: the Inverse Kinematic Calculator and the Motion Profile Generator.

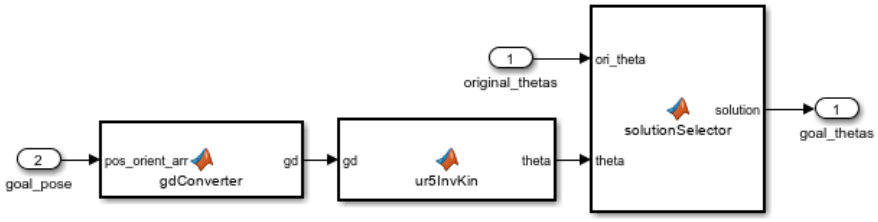


Figure 4 UR5 inverse kinematics solving algorithm block

As described in *Figure 4*, the calculation of goal_thetas (desired joint angles) begins with the goal_pose which specifies the desired position and orientation of the robot's end-effector. The gdConverter processes the input pose to the 4x4 homogeneous transformation matrix for the ur5 inverse kinematic calculator. A maximum of 8 possible joint angle solutions can be calculated using a geometric algorithm, as described below. These solutions are then refined in the solutionSelector block, where the optimal solution is selected based on criteria such as minimizing joint movement or avoiding singularities, using the original_thetas (last thetas or the initial thetas when starting the cycle) as a reference. The final output, goal_thetas, represents the joint angles required to achieve the specified pose.

Algorithm 1. inverse kinematic for UR5

Input:
Transformation Matrix T_0^6 (4X4)
DH Parameters

Output:
Joint angles array (1X6) (max. 8 possible solutions)

Calculate Theta1 (Joint 1 Angle)

Compute position vector p05 from gd and the robot geometry.
Compute intermediate angles (psi, phi) using trigonometric functions.
Use these angles to determine two possible values for theta1.

Calculate Theta5 (Joint 5 Angle)

For each possible theta1:
Compute the relative transformation T16 and position vector p16.
Extract the z-component of T16's position.
Solve for two possible values of theta5 using the z-component and d4 using trigonometric functions.

Calculate Theta6 (Joint 6 Angle)

For each combination of theta1 and theta5:
Compute the relative transformation T61.
Extract relevant rotation components of T61.
Solve for theta6 using trigonometric relationships.

Calculate Theta3 (Joint 3 Angle)

For each combination of theta1, theta5, and theta6:
Compute the relative transformation T14.
Extract the position vector p13 from T14.
Solve for two possible values of theta3 using the geometry of the arm.

Calculate Theta2 and Theta4 (Joint 2 and Joint 4 Angles)

For each combination of theta1, theta3, theta5, and theta6:
Compute the position vector p13's norm.
Solve for theta2 using trigonometric relationships involving p13.
Compute the relative transformation T34.
Solve for theta4 using rotation matrix components.

Adjust and Normalize Joint Angles

Adjust theta1 to remove offset.
Normalize all joint angles to be within the range [-pi, pi].

Return Results

Return the theta matrix containing 8 possible solutions for the joint angles.

The above algorithm is implemented adapted from Hawkins' work.[3]

Joint	a	α	d	θ
1	0	$\pi/2$	0.089159	θ_1
2	-0.425	0	0	θ_2
3	-0.39225	0	0	θ_3
4	0	$\pi/2$	0.10915	θ_4
5	0	$-\pi/2$	0.09465	θ_5
6	0	0	0.0823	θ_6

Table 1 D-H parameters

The second algorithm is the Motion Profile Generator, detailed below. This algorithm is used to create trapezoidal or triangular motion profiles for several reasons. It ensures that maximum velocity and acceleration limits are not exceeded, protecting mechanical components from stress and damage. These profiles provide smooth transitions by eliminating abrupt changes in velocity and acceleration, resulting in continuous and fluid motion. This precision in controlling acceleration, cruising, and deceleration phases enhances accuracy in reaching target positions. Additionally, smooth motion reduces mechanical wear, increases energy efficiency, and enhances safety, particularly in collaborative environments. Trapezoidal profiles are ideal for applications

requiring steady motion at constant speeds, while triangular profiles are suited for shorter or more constrained movements.

Algorithm 2. Motion profile generator for the motors

Input:
Maximum velocity (maxVel), maximum acceleration (maxAcc), target position (targetPos)

Output:
Position (Pos), velocity (Vel)

```
If target_position has changed:
    Determine the sign of motion from current position to target position (if need braking):
    If sign < 0:
        calcuate the braking time and distance to 0 velocity.
    Else:
        No braking is needed.
        calculate the acceleration time and distance accordingly.
Calculate total distance to target:
    distanceTotal = |targetPos - oldPos + sign * distanceBrake|
    Use distanceTotal, maxVel,  maxAcc to calculate the shape of motion profile.
Determine profile shape:
    If timeConstVelocity > 0:
        set shape = 1 (trapezoidal)
    Else:
        shape = 0 (triangular)
        Recalculate top velocity (topVel), acceleration, and deceleration times/distances for triangular motion.
Calculate the time elapsed since the last update:
    deltaTime = time - oldTime
Generate motion profile:
    Compute pos and vel based on deltaTime
```

Figure 5 illustrates the motion generator's velocity profile, showcasing both position and velocity dynamics. In the top plot, the orange line represents the actual position, which closely follows the commanded position depicted in blue, exhibiting a smooth S-shaped trajectory during transitions. The bottom plot displays the corresponding velocity profile, where a combination of trapezoidal and triangular profiles is employed. Trapezoidal profiles ensure steady motion at constant speeds, while triangular profiles are utilized for shorter or more constrained movements. Together, these profiles highlight the system's ability to optimize motion dynamics and ensure precise tracking of the reference position.

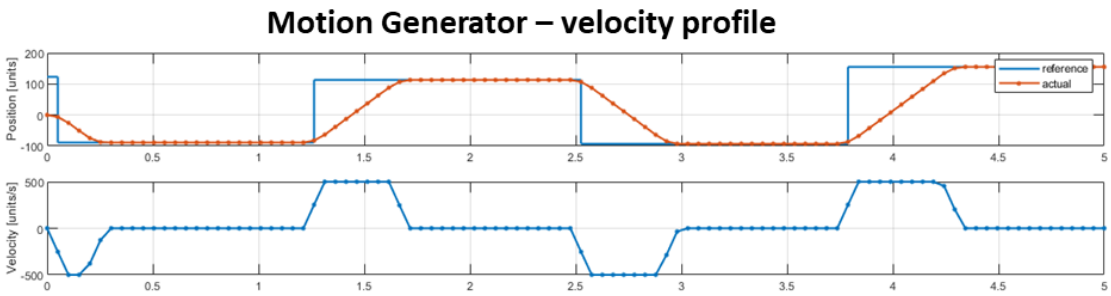


Figure 5 Position and velocity profile generated by the motion profile generator given the reference input signal

Each joint in the robotic system is managed by an independent controller comprising a Motion Generator and a DC Motor System as shown in *Figure 6*. The Motion Generator calculates the required motion parameters, such as position, velocity and acceleration, employing a trapezoidal or triangular motion profile for smooth transitions as described above. The DC Motor System executes the generated motion profile, ensuring precise and fluid joint movement.

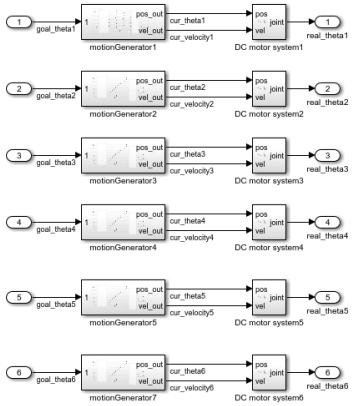


Figure 6 UR5 motion profile generator block for 6 different joints

DC Motor System incorporates a transfer function which is used to model the dynamic behavior of the DC motors as depicted in *Figure 7*. A feedback loop continuously monitors the current joint state, feeding it back into the controller to correct any deviations, thereby ensuring accurate, stable, and efficient motion control.

(a)

$$G(s) = \frac{K_m}{L_a J s^2 + (R_a J + L_a c)s + (R_a c + K_b K_m)}$$

Km	the motor torque constant
La	the inductance of the motor
J	The moment of inertia of the rotor
Ra	The resistance of the motor
La	The inductance of the motor
c	Motor friction constant
Kb	The velocity constant

(b)

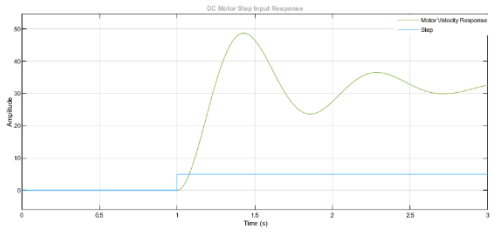


Figure 7 (a) Open Loop transfer function of a DC motor. (input is the applied voltage and output is the motor velocity) [4] (b) DC motor transfer function response (rad/s) with a given step input voltage (V)

The cur_velocity signal is calculated from the Motion Generator and is used to adjust the voltage supplied to the DC motor as shown in *Figure 8*. This adjustment modifies the motor's torque, enabling it to achieve the desired velocity. The real-time velocity signal is obtained from the motor and fed back into the control loop. A PID controller is implemented to minimize the error between goal velocity (cur_velocity) and the actual velocity, ensuring precise speed control. Additionally, the real-time joint angle (feedback_thetas) signal is fed into the loop to ensure the joint achieves the desired position (cur_thetas). The DC motor's dynamics are modeled using a transfer function block, which captures its response characteristics. To sum up, the DC motor system adjusts the motor's movement based on the generated motion profile, maintaining control over both the desired joint angle and velocity.

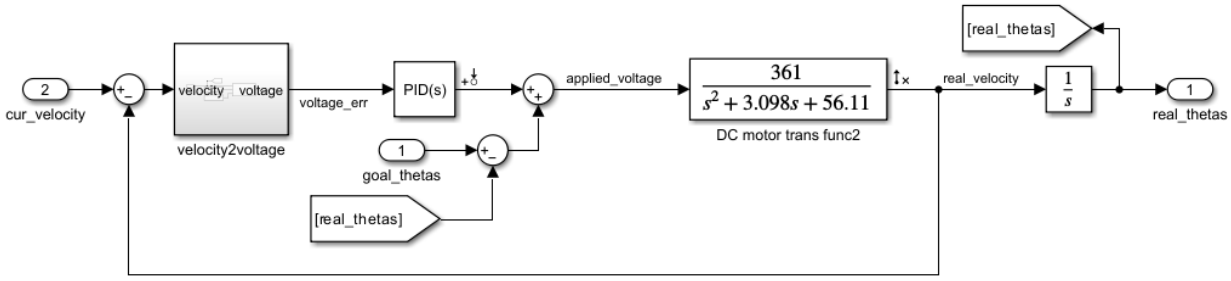


Figure 8 PID controller for the joint velocity and position with regarding to the generated motion profile

CONTROL SYSTEM ANALYSIS

The simulation results of the motion controller are demonstrated across several figures, showcasing its performance and accuracy. *Figure 9* (a) depicts the input signal to the model, an impulse signal with an amplitude of 0.1 and a 20% duty cycle applied as the x position input. *Figure 9* (b) illustrates the velocity, position, and acceleration profiles, where the maximum velocity and acceleration are configured to 180 rad/s and 300 rad/s², respectively. In *Figure 9* (c), the real end-effector x position aligns closely with the input signal, highlighting the controller's accuracy in tracking the desired motion profile. The end effector y, z position plots and rx, ry, rz orientation plots can be found in Appendix. *Figure 9* (d), (e), and (f) present the joint position, velocity, and acceleration profiles, respectively. The initial joint angles are equally zero. Notably, the motion profile generator ensures smooth, continuous, and differentiable velocity and acceleration curves, effectively preventing any spikes and demonstrating the controller's ability to maintain stable and precise joint dynamics. Amplified joint velocity and acceleration curves can be obtained from the Appendix.

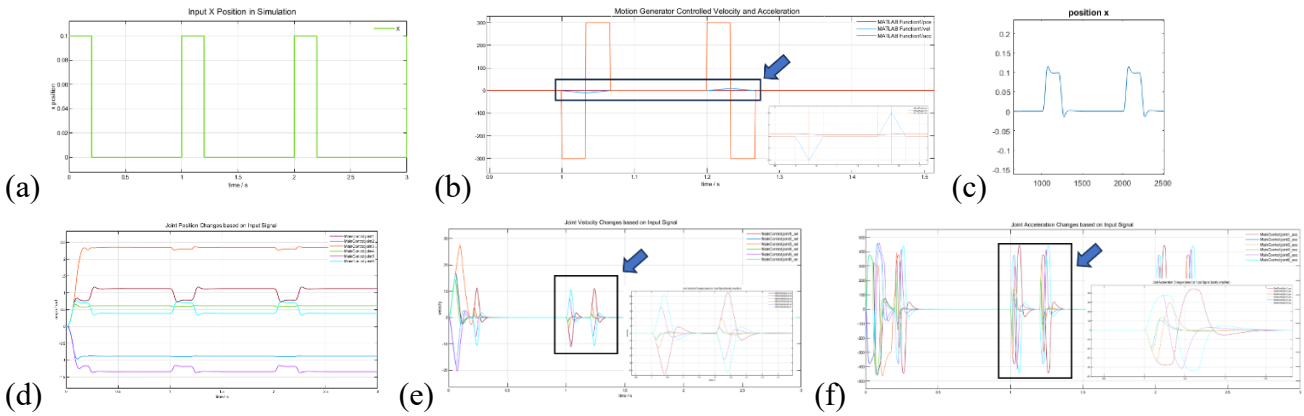


Figure 9 (a) Impulse input signal to the Simulink simulation with an amplitude of 0.1 and a 20% duty cycle. (b) Motion profile generated by motion profile generator describing position, velocity and acceleration changes. (c) X position changes at end-effector in response to the input signal. (d) 6 joint position changes in response to the input signal. (e) 6 joint velocity changes in response to the input signal. (f) 6 joint acceleration changes in response to the input signal.

The Motion Controller under analysis demonstrates robust performance in both the time and frequency domains, supported by the control latency analysis, Bode plot and Nyquist plot.

In the time domain, as depicted in *Figure 10* (a), the end effector's x-position achieves its target with impressive speed, stabilizing within 100 milliseconds. This rapid response indicates an efficient and well-tuned controller that minimizes overshot and settling time, critical for precision tasks in robotic systems. The 13% overshot observed in the PID control is attributed to the relatively large distance to the setpoint of 0.1 m. However, in practice, the setpoint distance is likely smaller due to the controller's high sample rate and physical limitations.

In the frequency domain, the Bode plot in *Figure 10* (b) reveals valuable stability insights. The measured phase margin is 11° at the 0 dB crossover frequency, reflecting the system's ability to maintain stability despite small disturbances or modeling uncertainties. The gain margin, exceeding 30 dB, indicates the system can tolerate substantial increases in loop gain before losing stability. The Nyquist plot in *Figure 10* (c) further confirms this stability; it does not encircle the critical point (-1,0), and the magnitude at the -180° phase frequency is below unity. These characteristics validate that the system adheres to stability criteria, ensuring consistent and reliable operation.

Overall, the robotic system is stable, with a rapid transient response and robust frequency-domain performance, making it suitable for high-precision applications in dynamic environments.

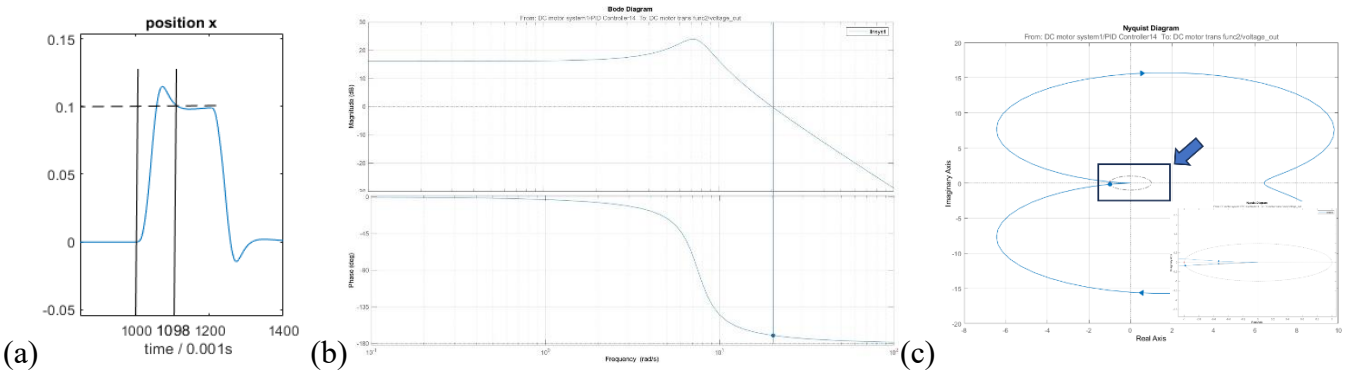


Figure 10 (a) X position plot in analysis of controller's performance in settling time and overshoot. (b) Bode diagram of the controller. (c) Nyquist diagram of the controller.

ROS2 AND CONNEXT

The diagram highlights the distinction between applications utilizing the ROS 2 API and those leveraging the DDS API, emphasizing the role of key framework components. At the core of the ROS 2 API are libraries like rclcpp, rclpy, and rcljava, which provide high-level abstractions for application development in C++, Python, and Java, respectively. These libraries simplify programming by managing nodes, topics, services, and communication interfaces. Beneath these layers lies the ROS Middleware (RMW), which acts as an abstraction layer over DDS implementations, enabling interoperability across various DDS vendors, such as RTI DDS, Fast DDS and Cyclone DDS. This layered structure introduces some latency, differentiating applications using the ROS 2 API from those using the DDS API directly. The experiment below is to measure these latencies in the applications of ROS2 and RTI DDS.

Applications leveraging the DDS API bypass the ROS2 API to communicate directly with the DDS Databus. This approach eliminates the additional latency introduced by the ROS 2 framework while maintaining full compatibility with ROS 2 data types and communication models as demonstrated in *Figure 11*. The DDS Databus serves as the core communication layer, supporting publish/subscribe (pub/sub) interactions and ensuring fully interoperable data exchange. While the ROS 2 API is ideal for rapid development and abstraction, the DDS API is well-suited for performance-critical systems requiring minimal communication overhead, making both approaches valuable depending on the application's requirements.

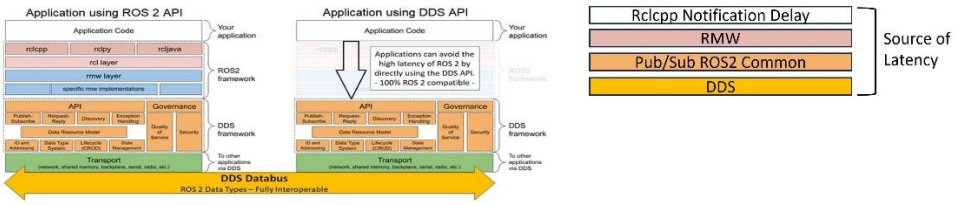


Figure 11 Comparison of application layers and latency sources in ROS 2 and DDS frameworks. [2][5]

In this experiment, up to 10 nodes are spawned and connected in a cascading manner, forming a pipeline through which messages are transmitted sequentially. Each node in the middle of the chain acts as both a subscriber and a publisher: it receives a message from the previous node, processes it, and then passes it to the next node. The publisher at the start of the chain appends a timestamp to the message content, allowing each subscriber to calculate the latency by subtracting the timestamp of the message sent by the previous node from the time it was received.

The communication uses a reliable transmission protocol, which ensures message delivery and ordering. This protocol is commonly used in both ROS 2 and RTI DDS environments, making the experiment representative of real-world applications. This setup allows for the measurement and analysis of end-to-end latency across the chain, helping to evaluate the performance of the messaging system under a reliable transmission protocol as shown in *Figure 12*.

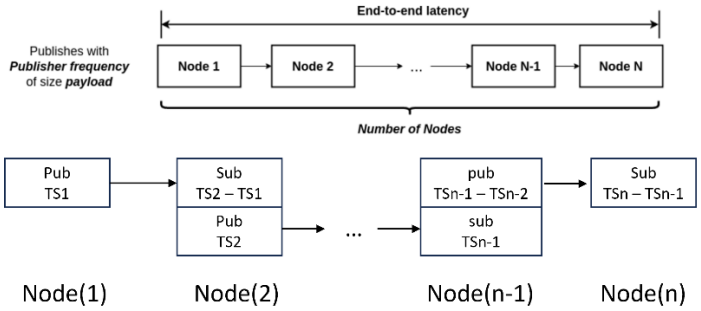


Figure 12 Experiment topology to analyze the end-to-end latency across multiple nodes in a publisher-subscriber framework.[5]

The experiment parameters as shown in *Table 2* include publisher frequencies of 1, 50, 100, 500, and 1000 Hz, payload sizes of 128b, 512kb, and 1M, and up to 10 nodes (2-10). The types tested are ROS 2 and Connext, using a reliable protocol for DDS. The environment is RTI DDS 7.3.0 and ROS 2 Humble. The experiment involves two distinct topologies to evaluate communication performance. In the first topology, all nodes are spawned on a single device (Device1), utilizing the SHMIO protocol for fast and efficient intra-device communication. In the second topology, nodes are distributed across two devices (odd nodes are spawned on Device1 and even nodes are spawned on Device2. More detailed can be found in *Figure 15*), with data transmission occurring over a network switch using the IPv4 UDP protocol to facilitate inter-device communication. These two setups represent commonly used configurations for evaluating communication performance.

Publisher Frequency / Hz	1, 50, 100, 500, 1000	**Environment:** RTI DDS: 7.3.0 ROS2: humble **Device1:** Processor: Intel(R) Core(TM) i7-14700KF 3.40 GHz. RAM: 32.0 GB. **Device2:** Processor: Intel(R) Core(TM) i7-4712HQ 2.30 GHz. RAM: 16.0 GB. **Switch:** TP-link TL-SG105E Gigabit switch
Payload	128b, 512kb, 1M	
Number of Nodes	2, 3, 4, 5, 6, 7, 8, 9, 10	
Type	ROS2, Connext	
Reliability (only for DDS)	Reliable	

Table 2 Experiment variables, software environment and hardware configurations.

LATENCY COMPARISION BETWEEN ROS2 AND CONNEXT

The latency analysis of the Connext and ROS2 applications under the first topology, where all nodes are spawned on a single device (Device1) utilizing the SHMIO protocol, reveals notable trends. Across all payload sizes (128B, 512KB, and 1MB), Connext generally exhibits lower latencies compared to ROS2, indicating its advantage in leveraging the SHMIO protocol for intra-device communication. In *Figure 13*, as payload sizes increase, there is a clear trend of rising latency, which is consistent with the added overhead of handling larger data volumes. Conversely, increasing the frequency of data transmission tends to decrease latency, likely due to optimizations in resource utilization and communication scheduling at higher frequencies. This pattern holds across both Connext and ROS2.

In *Figure 14*, the cumulative latency results for Connext and ROS2 applications across different nodes demonstrate that Connext consistently outperforms ROS2 in terms of lower latency. The data shows the same pattern that as the payload size increases, the cumulative latency also increases for both systems, but Connext maintains a significant advantage in most cases. This indicates its superior handling of intra-node communication and payload efficiency. Additionally, the trends highlight that Connext performs particularly well at higher frequencies (e.g., 1 kHz), showcasing its ability to maintain low latency under high-demand scenarios. This reinforces the reliability and efficiency of the Connext application in minimizing latency across diverse payload and frequency combinations. These insights underscore the critical role of protocol implementation in optimizing latency for high-frequency, large-payload communication scenarios.

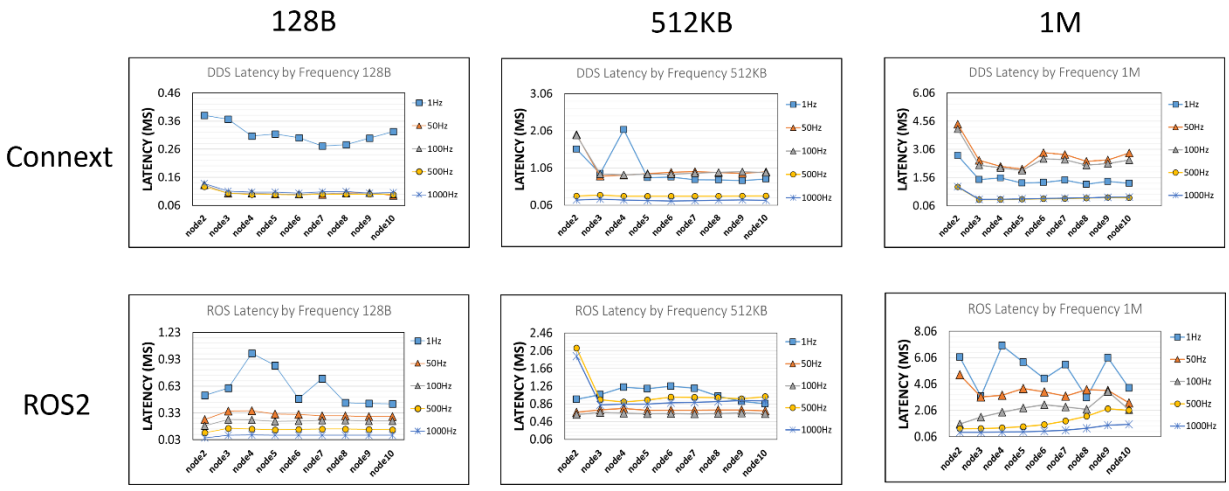


Figure 13 Investigation of latency across a scalable node system with different transmission frequency and payload

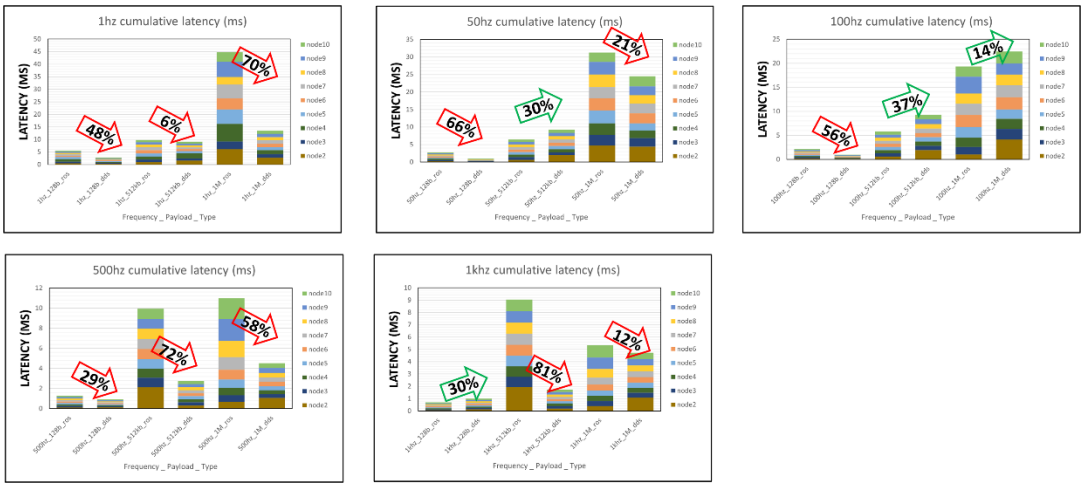


Figure 14 Investigation of cumulative latency of a node system with different transmission frequency and payload

Under the second topology, where nodes are distributed across two devices ("aliensteve" and "justice"), as shown in *Figure 15*, with data transmission facilitated by the IPv4 UDP protocol over a network switch, the latency analysis of Connext and ROS2 applications reveals distinct patterns. The inter-device communication introduces additional latency compared to the first topology due to network overhead and transmission delays. The figure highlights the clear separation of nodes between the two devices, with data transmission occurring back and forth between them. This setup tests the ability of the protocols to handle inter-device communication efficiently.

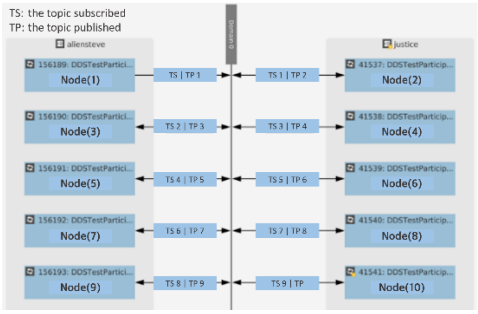


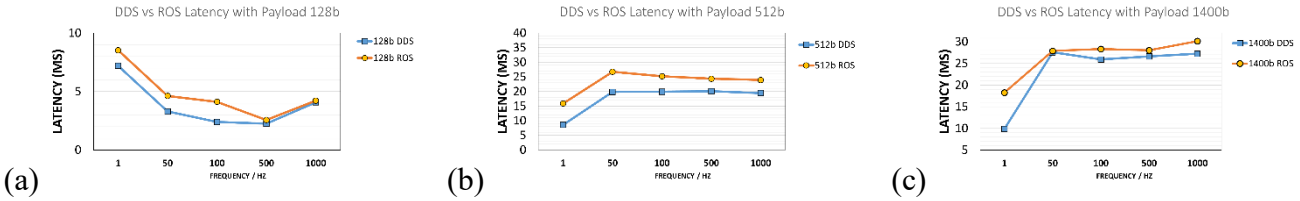
Figure 15 Active DDS communication overview from RTI Console: device names, topics, publishers, and subscribers.

To address the challenge of unsynchronized clocks across devices, the latency is calculated using timestamps TS9 − TS1, TS9 and TS1 both recorded on the same device. This method ensures accuracy and eliminates errors caused by clock discrepancies between devices. Payload sizes are limited to 128B, 512B, and 1400B to avoid the complexities and variability introduced by UDP

fragmentation with larger payloads. This approach maintains precision by leveraging *std::chrono::high_resolution_clock::now()* and focuses on payloads within the network's maximum transmission unit (MTU), providing consistent and reliable latency measurements without relying on external synchronization methods like NTP, which may add latency overhead.

$$sum\ of\ latency = TS2 - TS1 + TS3 - TS2 \dots \dots + TS9 - TS8 + TS10 - TS9$$

The test results as shown in *Figure 16* reveal that latency increases consistently with larger payload sizes (128B, 512B, 1400B) in both the Connext and ROS2 applications, as shown in the respective figures. This aligns with the established understanding that higher payloads lead to greater latency due to the additional data processing and transmission overhead. Notably, the Connext application outperforms ROS2 across all payload sizes and frequencies, demonstrating superior efficiency and reliability. These findings reinforce the advantage of Connext in scenarios requiring lower latency and high-performance communication.



(a) (b) (c)

Figure 16 Comparison of latency between DDS and ROS in different frequency and the UDP transmission protocol with different payloads of (a) 128b (b) 512b (c) 1400b.

REFERENCES

[1] Model-Based Design - MATLAB & Simulink
[2] Quigley, M., Gerkey, B., & Smart, W. D. (2021). Micro-ROS. In *Robot Operating System (ROS): The Complete Reference (Volume 7)* (pp. 45–70). Springer. https://doi.org/10.1007/978-3-031-09062-2
[3] Hawkins, K. P. (2013). Analytic inverse kinematics for the universal robots UR-5/UR-10 arms.
[4] Ali, A. M., Mohammed, A. H., & Alwan, H. M. (2019). Tuning PID Controllers for DC Motor by Using Microcomputer. *Int. J. Appl. Eng. Res*, *14*(1), 202-206.
[5] Kronauer, T., Pohlmann, J., Matthé, M., Smejkal, T., & Fettweis, G. (2021, September). Latency analysis of ros2 multi-node systems. In *2021 IEEE international conference on multisensor fusion and integration for intelligent systems (MFI)* (pp. 1-7). IEEE.

APPENDIX

As shown in *Figure 17*, the GUI design for the robotic system's movement simulation features a split-view interface, divided into four quadrants. The primary focus is the third quadrant, which provides a top-down view of the robot's movement in the x-y plane, capturing real-time positional data of end effector in response to the touchpad inputs. This quadrant is essential for observing and analyzing the robot's intended trajectory. The second and fourth quadrants offer detailed insights into potential fluctuations or inaccuracies in the robot's motion, with a focus on variations in the x and y axes, as these are the only variables in the target pose. The first quadrant presents a 3D visualization of the robot's movement, enabling users to assess spatial accuracy and overall trajectory in a three-dimensional environment. The movement window simulates a 300mm square, approximating the size of the abdominal cavity to replicate real-world surgical constraints, ensuring practical relevance and precision during development and testing.

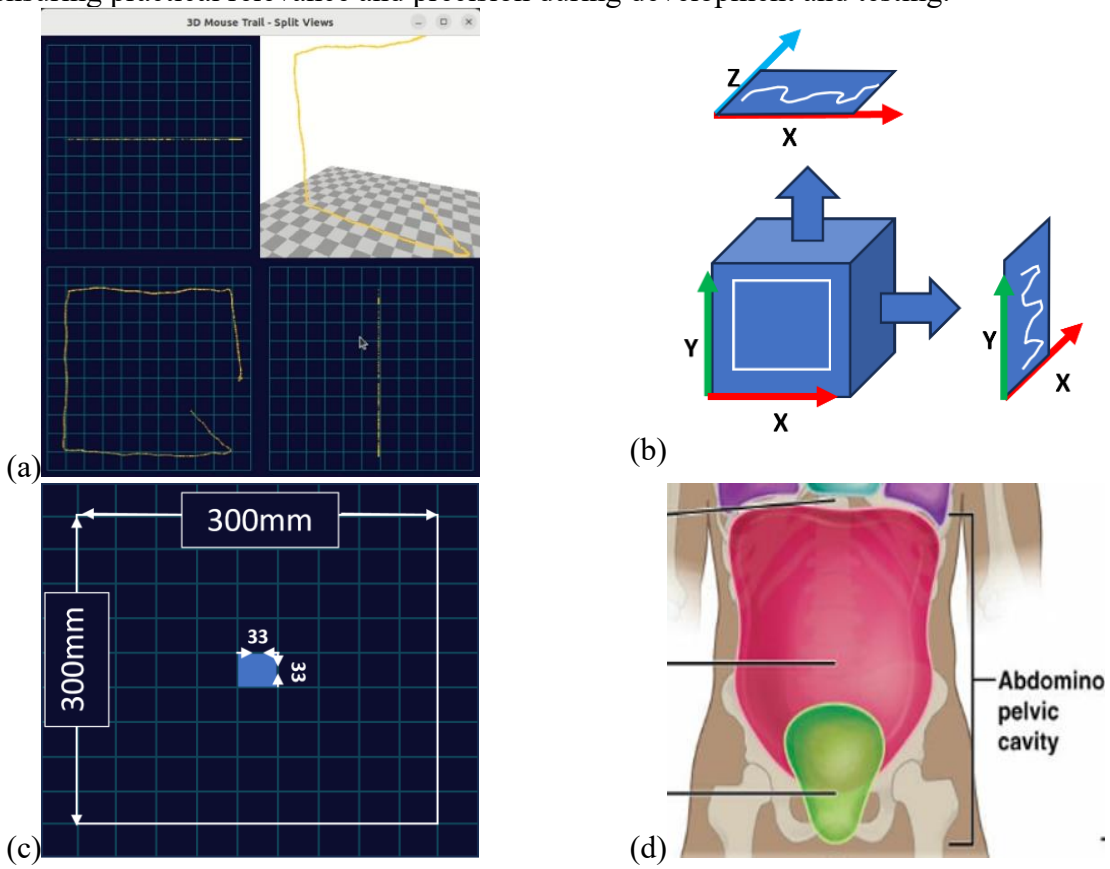
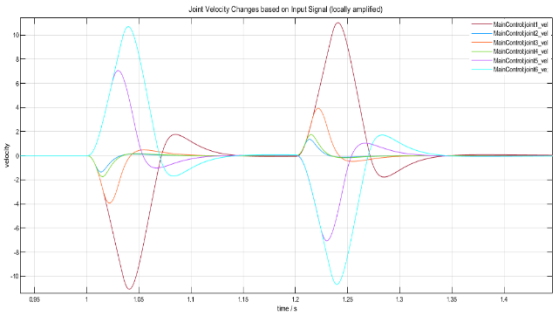


Figure 17 (a) The overview of the application GUI in four quadrant design. (b) An explanation to the view planes to quadrant 2, 3, and 4. (c) Grid representation of a 300mm x 300mm workspace of the touchpad area. (d) Abdominal cavity schematics.

As shown in *Figure 18*, the amplified plots of joint velocity and acceleration in the Simulink simulation demonstrate that the system operates within the predefined limits of 180 rad/s for velocity and 300 rad/s² for acceleration, ensuring the robotic joints remain within safe and efficient operating ranges. The curves are smooth and continuous, reflecting an absence of sharp changes in both the first-order and second-order derivatives for all six joints. This smoothness provides

several advantages: it minimizes mechanical stress on components, thereby reducing wear and extending the lifespan of the system. Additionally, it enhances energy efficiency by avoiding abrupt power surges.

(a) 

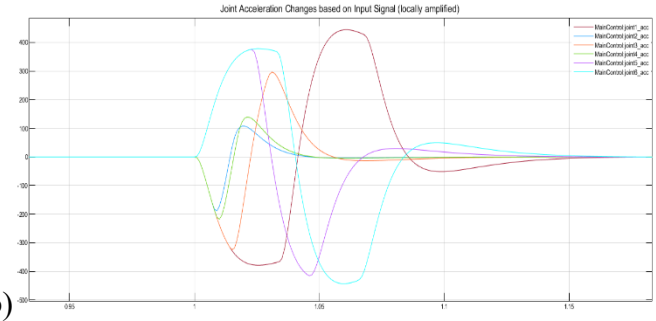
(b) 

Figure 18 (a) Amplified joint velocity in response of input signal. (b) Amplified joint acceleration in response of input signal

As shown in *Figure 19*, the six plots presented below illustrate the dynamic changes across six dimensions in the Simulink simulation, capturing the system's behavior in terms of x, y, z position and Rx, Ry, Rz of robotic end effector.

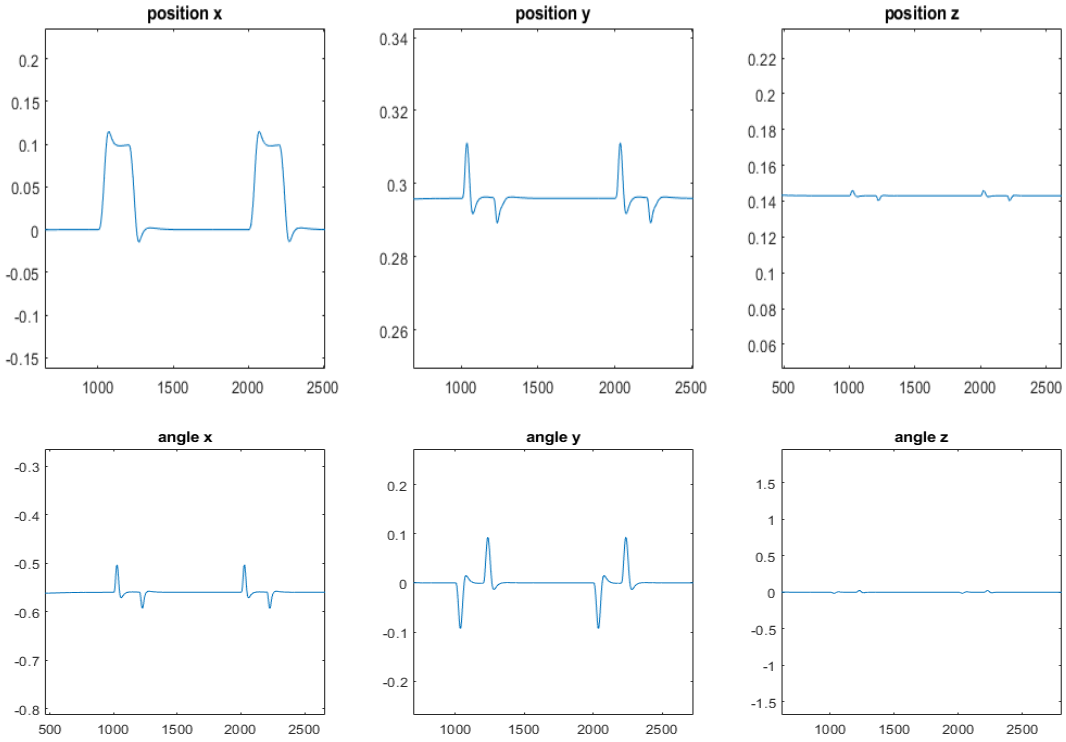


Figure 19 End-effector position and orientation changes over time in response of input signal: x, y, z positions and Rx, Ry, Rz orientations

CODE

The codes for the application and experiments can be found on GitHub at the following link: [https://github.com/stevenleon99/DDS_URControl].