2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Macau, China, November 4-8, 2019

# An Asynchronous Multi-Body Simulation Framework for Real-Time Dynamics, Haptics and Learning with Application to Surgical Robots

Adnan Munawar and Gregory S. Fischer

*Abstract*— **Surgical robots for laparoscopy consist of several patient side slave manipulators that are controlled via surgeon operated master telemanipulators. Commercial surgical robots do not perform any sub-tasks – even of repetitive or non-invasive nature – autonomously or provide intelligent assistance. While this is primarily due to safety and regulatory reasons, the state of such automation intelligence also lacks the reliability and robustness for use in high-risk applications. Recent developments in continuous control using Artificial Intelligence and Reinforcement Learning have prompted growing research interest in automating mundane sub-tasks. To build on this, we present an inspired Asynchronous Framework which incorporates real-time dynamic simulation – manipulable with the masters of a surgical robot and various other input devices – and interfaces with learning agents to train and potentially allow for the execution of shared sub-tasks. The scope of this framework is generic to cater to various surgical (as well as non-surgical) training and control applications. This scope is demonstrated by examples of multi-user and multi-manual applications which allow for realistic interactions by incorporating distributed control, shared task allocation and a well-defined communication pipe-line for learning agents. These examples are discussed in conjunction with the design philosophy, specifications, system-architecture and metrics of the Asynchronous Framework and the accompanying Simulator. We show the stability of Simulator while achieving real-time dynamic simulation and interfacing with several haptic input devices and a training agent at the same time.**

## I. INTRODUCTION

Partial autonomy of sub-tasks has exciting prospects for research aimed for the next generation of surgical robotics. Research in this area focuses on assisting the surgeon in accomplishing sub-tasks, thereby making the automation passive in nature. Some notable research in this area includes autonomous algorithms for performing soft-tissue suturing [1], an automated approach for sinus surgery using computer navigation techniques [2], characterization and automation of soft-tissue suturing using a curved needle guide [3] and automation of cutting/creasing sub-tasks while employing learning by observation [4]. Additionally, [5] presents a holistic approach to simplifying the task of manipulator positioning prior to surgeon interaction, and [6] demonstrates a telemanipulated surgical simulation designed for heart surgery. A trainable infrastructure is presented in [7] with controllable dominance and aggression factors for automating repetitive surgical tasks. Lastly, a shared infrastructure for collecting da Vinci Research Kit (dVRK) manipulators and vision data, primarily for training learning agents by motion decomposition of sub-tasks is developed in [8].

Recent developments in deep learning and AI have sparked the interest of researchers from a variety of fields. Until very recently, the scope of deep learning algorithms were limited to discretized problems, and thus, most real world control problems remained out of reach. However, the introduction of Deep Deterministic Policy Gradients (DDPG) model [9] – an improvement over Deterministic Policy Gradients – broke new grounds while making the realization of smart agents for continuous control problems seemingly possible. These advancements led to the successful training of a simulated human rag-doll capable of running, jumping and avoiding obstacles [10]. Unsurprisingly, there has been an increase in the number of software libraries targeted for machine and reinforcement learning developed by the open source community. Many of these libraries provide Python API's and are capable of utilizing high-bandwidth system resources for faster training of data. Zamora et al. [11] present a useful reinforcement learning toolkit catered towards mobile robots that employs such a Python interface for reinforcement learning by interconnecting the Gazebo simulator with OpenAI's GYM [12].

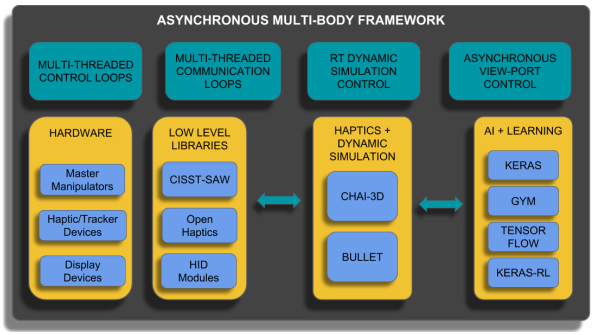



Fig. 1: An overview of components selected for the Asynchronous Framework for Assistive intelligence, simulation and collaborative control.

We propose the use-case of an Intelligent Agent for collaborative control of real-time tasks, specialized for robotic surgery. Given that we intend to assist the Master with collaboration rather than fully automatic control, we use the more appropriate term of Assistive Intelligence. The coordination can range between two target types, (1) multi-sensory feedback to the Master – visual, haptic and tactile – and (2) cooperative control of one or more slaves in conjunction with user-controlled manipulators. We propose a framework to achieve both forms of assistance by providing

Adnan Munawar & Gregory S. Fischer are with the Department of Robotics Engineering, Worcester Polytechnic Institute, MA, 01609, USA [amunawar, gfischer]@wpi.edu

This work is supported by the National Science Foundation through National Robotics Initiative Grant (NRI): **IIS-1637759**

the means to integrate modern surgical robots/haptics devices with high fidelity asynchronous control, haptic feedback and the implementation of a distributed asynchronous framework for manipulating simulated dynamic bodies that allow for the training of learning agents.

## II. High Level System Architecture

An essential step towards the realization of the proposed Asynchronous Framework is the stable and robust control of multiple input devices in the simulated dynamic environment. Section III-A discusses the challenge associated with this multi-device control. The second, but equally important, step is the selection of the right software components which is driven either by the compatibility in our use-case (CISST-SAW and dVRK – an open-source research kit based on the clinical da Vinci surgical robot – [13] [14]) or the popularity and adoption of the components in the research community. As such, the existing components which have been chosen to complement the Asynchronous Framework include CHAI-3D [15], Bullet [16], Keras [17], Keras-RL [18], and OpenAI's GYM [12]. Figure 1 shows a holistic view of the inclusion of the aforementioned components in Asynchronous Framework.

The motivation behind the selection of each component is presented briefly, starting with the two integral components, Bullet [16] and CHAI-3D [15]. Bullet's Dynamics Engine is already used in some open-source robotics simulators, including Gazebo - the preferred dynamic simulator for the Robot Operating System (ROS) community. While Open Dynamics Engine (ODE) is another competitive physics library, Bullet provides a built-in collision detection library. CHAI-3D is an open-source library that supports a vast majority of commercial haptic devices and offers a device agnostic interface to applications rendered in Open-GL [19]. CHAI-3D lacks a built-in physics computation library but has preliminary support for modules built around Bullet and ODE.

Keras [17] is chosen because of its compatibility with modern libraries for training Neural Networks, and its ease of use. Keras-RL [18] is built to support Keras and provides the implementations of various Reinforcement Learning algorithms. OpenAI's GYM allows for the creation of environments and agents that expose an action-state interface for input-output and is the default frontend for utilizing Keras-RL (and consequently Keras for training Neural Networks using TensorFlow).

The Asynchronous Framework is realized in an application called the Asynchronous Multi-Body Framework (AMBF) Simulator which provides a dynamic-haptic simulation (rendered by utilizing CHAI-3D's interfaces and Bullet as the dynamics solver) and allows for high-fidelity control via robust inter-process communication interfaces. The AMBF Simulator utilizes a ground-up design philosophy that allows for control of each dynamic object in an intuitive asynchronous fashion (Figure 2). The details, discussions, and metrics of the simulator are presented in Section III-B. Additionally, we present a Python Client (AMBF Client) that

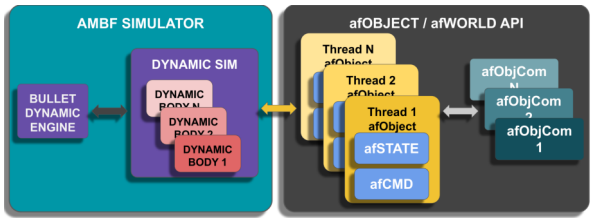



Fig. 2: A visual representation of the Asynchronous Framework with regards to the C++ AMBF Simulator where each simulated dynamic object is represented as an *afObject*. The *afObject*s utilize independent communication pipelines by exposing State/Command interfaces which allow isolated control

utilizes the communication interfaces exposed by the AMBF Simulator and complements it to provide robust and easy-to-use interfaces for training on real-time dynamic tasks with simulation in the loop haptic-feedback (Section III-E).

## III. Implementation Details

### A. Asynchronous Control of Multiple Haptic Devices

Bullet Physics is used as a module in CHAI-3D to simulate dynamic bodies while using haptic/input devices to interact with them. This built-in module is designed to interface a single haptic device while using a fixed time-step for the dynamic update. The fixed time-step allows for a stable performance, but it lacks accuracy as it does not track the real world clock. A realistic multi-manual task requires multiple input devices interacting with each other and bodies in the simulation, all while maintaining a recommended haptic update-rate ($\geq 1kHz$) and the dynamic simulation clock in sync with the real-world clock. Achieving this setup is not trivial as the challenges are inherent to the implementations of rigid body dynamics using numerical integration and constraint solving methods. Regardless of the dynamic engine used, the overall simulation typically has an update-step in which it applies the forces/constraints/collisions to children objects and numerically integrates over the given time-step. In a sense, this update-step synchronizes all the dynamic bodies, and hence, all the body constraints require resolution prior to each step.

The computational time of each update-step depends on several internal, as well as external, (OS scheduling) factors. The internal factors are mainly the magnitude of the time-step $\delta t$ and the collision computation of high-density meshes during contact. To keep in sync with the real world clock, the time-step needs to be calculated simultaneously on each update-step. Adding several input devices to simulation adds to the computational time, thereby increasing the duration of successive time-steps. Getting bounds on the time-step is not trivial in a non real-time OS and unbounded time-steps lead to cyclic deterioration.

Each haptic device is represented by a simulated dynamic end-effector (SDE) and is controlled using a dynamic control law. In a trivial implementation, all the devices are sequentially read to calculate and apply the action forces on their SDEs, next, the simulated world is stepped forward by $\delta t$ which updates the states of dynamic bodies. Finally, the

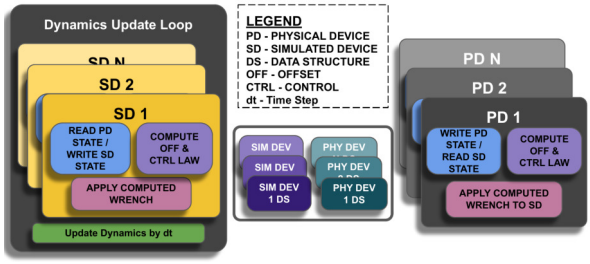



Fig. 3: A block diagram depicting the Design of Asynchronous Control Scheme, the Simulated end-effectors and Devices maintain independent and mutually exclusive Data Structures (DS) that are updated on successive writes and are capable of asynchronous reads

reactionary forces are applied to the corresponding input devices in the same time-step.

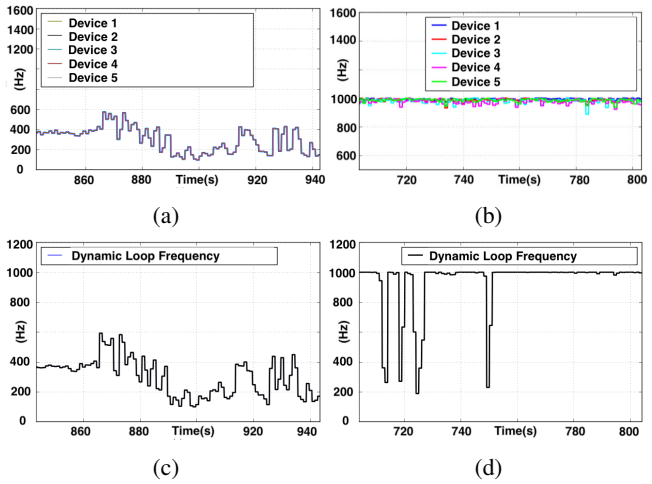



Fig. 4: Figure (a) and (b) show the haptic update-rate of 5 devices when controlled 'sequentially' vs 'asynchronously', respectively. Figure (c) and (d) show the corresponding rates for physics update-loops for 'sequential' vs 'asynchronous' control

Controlling each device in task-space requires a large number of matrix operations, including similarity transforms to enable hand-eye (camera) coordination and offset-transforms for clutch engaging/disengaging (presented in Section III-C and equation 4). The drivers for several commercial devices – Geomagic Phantom/Touch from *(3D Systems Corp, Rock Hill, SC, USA)* and Falcon (*Novint Technologies Inc., NY, USA*) – impose a delay while commanding forces to restrict the update-rate. Tracker devices (such as Razer Hydra from (*Razer Inc., CA, USA*) operate at lower update-rates ($\leq 400Hz$), and hence pose additional challenges. Therefore, the issue with the "sequential" implementation is that reading/writing multiple devices throttles the update-rate of the dynamic and haptic feedback loops. Moreover, mixing devices with different update-rates makes the dynamic simulation unusable. This can be alleviated by withholding the force commands in the main loop and executing them concurrently in a separate thread. However, while this improves the update-timing, it makes the devices and SDEs unstable due to a non-deterministic delay between the computation of control laws and the application of output forces.

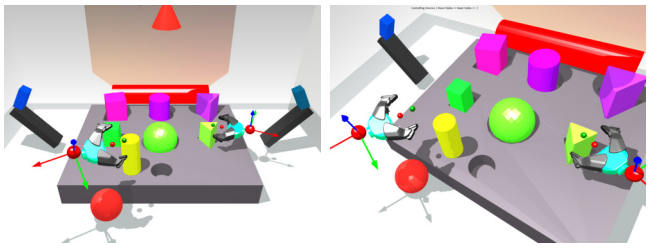
To counteract these issues, an asynchronous control scheme is implemented where the loop delays are isolated from each other to prevent cyclic deterioration. A block diagram representing this control scheme is shown in Figure 3. Implementation wise, the dynamic update-loop runs in a separate thread and all of the haptic update-loops in separate individual threads. Each input device owns a data-structure which is shared to allow for asynchronous reads and writes. This data-structure maintains the device's states and has fields to store the commanded forces. A similar, but non-identical, data-structure is defined for each SDE. The novelty in this implementation is the application of forces/commands in dynamic and haptic threads, as the execution counters of each thread are asynchronous by design. The difference between the two control schemes is analyzed by experimenting with a multi-manual task of grasping, picking and placing objects and recording the dynamic and haptic update-rates.

In one example configuration, the framework is stressed by simultaneously testing five haptic devices including two Novint Falcons, a Geomagic Touch, and two master telemanipulators (MTMs) from (*Intuitive Surgical Inc., Sunnyvale, CA, USA*). As shown in Figure 4(a), (c) in the sequential implementation, the update-rate never meets the 1 kHz set-point. On the other hand, in Figure 4(b), and (d), the device update-rates stay close to 1 kHz but the dynamic update-rate can swing depending upon the collision computation for high-density meshes during contact. The states and commands are stored outside the haptic/dynamic update-loops and are then used as "set-points" in the relevant threads to prevent saturating the forces in both simulation and haptic feedback loops.

### B. Design of Asynchronous Framework (AMBF) Simulator

We used a design philosophy, motivated by several different sources, which assimilates the concept of bodies in dynamic simulation as independent objects with self-contained kinematic & dynamic properties, thereby mimicking real-world objects. This philosophy is to distinguish from the practical implementation where the simulated bodies are part of an interconnected graph in a unified simulation and require sequential updates. The goal of this design philosophy is to allow for asynchronous manipulation and control of each simulated body independently. As a result, objects in the simulation are classified as either *afObject* or *afWorld*, where 'af' stands for 'Asynchronous Framework'. An *afObject* is a kinematic or dynamic rigid body which can have any number of movable parts (including 0). At their core both *afObject* and *afWorld* have two interfaces for communication, utilizing ***afState*** / ***afCommand*** for state / command pair. These two interfaces implement a generic input-output design that is easy to scale and communicate in parallel through an Inter Process Communication (IPC) medium (Figure 2).

The AMBF Simulator has a single world instance which is responsible for managing all the visual, kinematic and dynamic objects. This world instance supports features such as step-throttling, step-skipping and reporting metrics (discussed in more detail in Section III-F).



(a) (b)

Fig. 5: These figures show the simulated end-effectors controlled by dVRK Master with clutch/camera foot-pedals enabled. The clutch is used to move the haptic device disengaged, and the camera foot-pedal is used to re-orient the view-direction without affecting the end-effector

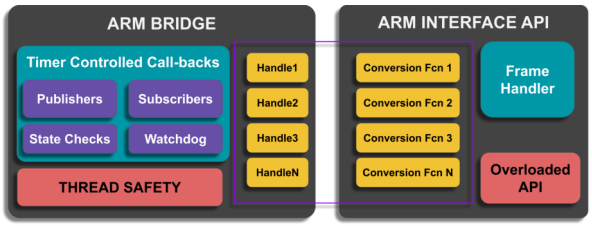



Fig. 6: This block diagram depicts a plugin based interface for dVRK manipulators using ROS as an IPC. The ROS functionality is sealed in the Arm Bridge Class whereas the ARM Interface exposes API for user applications.

### C. Integration of Constraints

The AMBF Simulator may contain sets of constrained bodies connected via sliding or rotating joints. Additionally, realistic multi-manual simulations targeting surgical applications may require grippers, forceps or retractors as simulated dynamic end-effectors (SDEs). For such SDEs, the abstract control of the jaw angle is preferable over explicit joint position or effort control. As a result, rather than providing angular limits and independent joint control for the SDE's, we use a more straightforward range between 0.0 and 1.0 for fully closed and fully open SDE jaw position. Interpolation is used for the values in between.

Figure 5 shows a pair of SDEs which act as proxies for dVRK MTMs and are controlled using a modified PD control law $F_{sim} = [f; \eta]$, where:

$$f = K_l \delta x t_s + B_l \delta^2 x / dt_d \tag{1}$$

$$\eta = (K_a \delta \theta t_s + B_a \delta^2 \theta / dt_d) z \tag{2}$$

Here $f$ and $\eta$ are the force and torque, while $K$ and $B$ are Stiffness and Damping coefficients. The term $t_s = \frac{dt_f}{dt_d}$ enables us to scale the time-step for asynchronous control by taking the fraction of custom fixed time-step ($dt_f$) by the dynamic time-step ($dt_d$). The control law outputs a spatial wrench which is added at each dynamic update-step of the physics simulation. Since the output wrench is added to the existing external forces on a simulated body, $t_s$ prevents the saturation of external forces for slower update-rates. Conveniently, it is often the case that, $t_s = 1$ (such as the dynamic simulation running at intended speed). $\delta x$ and $\delta \theta$ are the linear and angular offsets, and $z$ is the axis of angular offset. These quantities are calculated from equations 3 and 5.

$$\delta x = x_s^{n-1} + R_c(x_h - x_h^{p+}) - x_g \tag{3}$$

$$R_s = R_s^{p-} R_c (R_h^{p+})^T R_h R_c^T \tag{4}$$

$$(z, \delta\theta) = AxisAngle(R_g^T R_s) \tag{5}$$

In equations 3 and 5, $R$ denotes the rotation matrix. The super-script for the rotation matrix is ignored when it is represented w.r.t. the world frame. Special use is made of the superscripts $p+$ and $p-$. $p+$ denotes the event when the clutch/camera button is pressed, and $p-$ denotes the release event. In this sense, $R_h^{p+}$ is the recorded rotation matrix when the camera/clutch button is pressed, whereas, $R_s^{p-}$ is the simulated rotation recorded when the camera/clutch button is released. The remaining subscripts are defined as follows: $l = linear$, $a = angular$, $s = simulated$, $h = haptic \quad device$, $c = camera$ and $g = gripper$.

The force feedback on the dVRK Masters is computed according to equation 6. The term $(J(q)^T)^\dagger \Upsilon(q, \dot{q}, \ddot{q}) \Pi_E$ is the estimated gravity wrench and is appended to the computed-wrench from the simulation. We have done previous work on the development of a haptic interface for the dVRK manipulators [20] for this purpose. The dVRK masters are interfaced using a plugin called **DVRK Arm** demonstrated in Figure 6 which abstracts the CISST-SAW [14] implementation. This plugin based interface fully supports the Asynchronous Framework and hides all ROS-functionality from the AMBF Simulator. The details of **DVRK Arm** are beyond the scope of the current manuscript, but the source code can be found at [21].

$$F_{mtm} = (J(q)^T)^\dagger \Upsilon(q, \dot{q}, \ddot{q}) \Pi_E + T_{sim}^{mtm}(-F_{sim}) \tag{6}$$

### D. The Communication Medium

We experimented with shared memory and sockets as IPCs in Linux. The complexity involved at the cost of communication speed was not justified in creating a scalable solution for shared memory. Socket communication, although relatively slower, is scalable and provides similar implementations across all dominant Linux flavors and even other operating systems and programming languages. It does, however, require data-serialization and de-serialization. In addition to enjoying substantial community support, ROS [22] has the ability for serialization and socket communication together and thus, made it an appropriate candidate for our Asynchronous Framework.

Each *afObject* (and the single instance of *afWorld*) utilizes a plugin for *afObj*Comm (*afWorld*Comm in *afWorld*'s case) while using ROS as a middleware. The communication plugins use the same thread as their owners, and unlike other solutions for independent nodes over a distributed network (ROS Nodelets `http://wiki.ros.org/nodelet`), the AMBF Simulator uses a single node and distributes/isolates the callbacks using custom callback queues. This isolation

provides the flexibility to launch the communication instances from within the AMBF Simulator without the need for ros-launch files.

Each communication instance owns a WatchDog timer which has a primary and a secondary function for data transmission control. The primary function of the Watchdog is to reset the *afCommand* if the timing condition – the invocation frequency of the *afObj*Comm/*afWorld*Comm callback – is not met. This keeps the asynchronous control safe for physical devices connected to AMBF Simulator. The Watchdog timer is re-initiated once a stream of new commands starts flowing in. The secondary purpose of the watchdog timer is to limit the publishing frequency of *afState*s to lower values if the watchdog timer expires, thus reducing the use of computing resources.

### *E. The Python Client*

As discussed in section I, many of the popular libraries for learning and training agents have Python interfaces (Keras, GYM, Tensorflow/Theano, and Keras-RL). In alignment with these preferred interfaces, we present a stand-alone Python client that complements the AMBF Simulator. This client is capable of creating callable instances of *afObject*s and *afWorld* (using ROS Communication) which are isolated from one another to reduce communication and computational overheads. We outlined various specifications that prioritize robustness in handling load. These design specifications are intended for real-time training on data as well as closed-loop control by accounting for the communication overheads and slower execution speeds of Python applications. Based on these specifications, the Python Client uses data sequencing techniques and payload time-stamps to keep track of states, actions and rewards. The consequence of the design specifications is reflected not only in the Python Client itself but also in the AMBF Simulator and the Payload Types (Section III-F). A block diagram representing the Python Client is shown in Figure 7.

For safety reasons, each callable instance of *afObect* and *afWorld* in the client inherits a WatchDog timer which enforces command resetting if the timing condition fails. The Python Client is capable of throttling the dynamic update-loop of the AMBF Simulator, in which case, it provides a clock to step the dynamic update-loop. This clock is provided using "Clock" field in the ***afCommand*** message for *afWorld* and **number of jump steps** can be set to $> 1$. All this is done automatically by the Python Client as the user/training agent sets the corresponding parameters at run-time.

### *F. Online Training and Control and Associated Challenges*

Based on the considerations in Section III-E, the Communication Payloads for online training and control are designed accounting for the communication overheads and the slower execution speeds of Python applications. In this regard, the contents of *afObject*s and *afWorld*s communication payload are shown in table I. The naming convention is designed to be self-explanatory, however, some fields pertaining to the scope of this manuscript are explained. The message field **Ena Throttle** is used to control the flow of simulation based on the toggle of **Clock**. The **Jump Steps** is the number of steps the simulation must take between each clock toggle (event). The requirement for throttling the simulation comes from the action-reward pair for the valid Markov States in RL problems. This requirement mandates the states to have associated rewards which are meaningless if the simulation is not throttled between the update-steps of training (forward and backward pass of the Neural Network). **Server Time** is the time (to nano secs precision) of the clock running in AMBF Simulator, and the **Sim Time** is the time of the inner-clock of dynamic simulation. This time is incremented at each iteration of the dynamic solver such that:

$$t_{sim}^{n} = t_{sim}^{n-1} + dt_d \tag{7}$$

Since the dynamic update-loop (**Dyn Freq**) runs asynchronously without any real-time constraints, using a fixed $dt$ causes time dilation between the wall (world) and simulation clock (shown in Figure 8a with one input device and dynamic time-step $dt = 0.001$). The reason for this dilation is evident. Lacking a real-time kernel and custom sleep function makes it harder to meet the desired frequency which shifts the two clocks. Even with a real-time kernel, the start-up time for initializing haptic devices can throw off the simulation clock. Moreover, the nature of collision computation techniques in physics simulation libraries makes the computational time variable, and in effect, non-deterministic as it depends on the varying number of bodies in contact and their geometries.

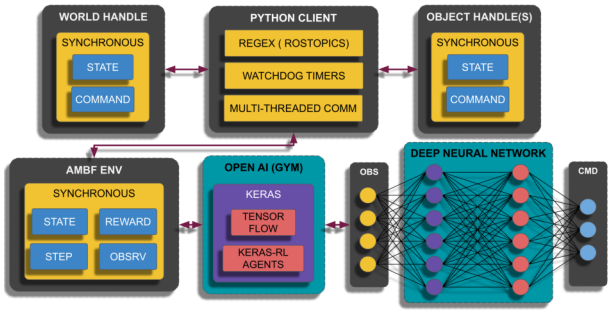
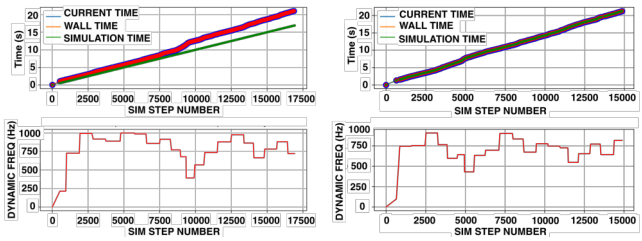



Fig. 7: The Python Client communicates with the AMBF Simulator using ROS as a middle-ware, AMBF ENV retrieves the requested handles for objects from Python Client and provides a GYM compatible interface




(a) (b)

Fig. 8: (a) Time dilation between Application Clock & Simulation Clock using fixed time-step (dt=0.001) (b) Time tracking between Application Clock & Simulation Clock using dynamic time-step

TABLE I: *afObject*s and *afWorld*s payloads for closed loop control and training via online data

| **afWorld** | | **afObject** | |
|---|---|---|---|
| **afState** | **afCommand** | **afState** | **afCommand** |
| - | - | Base Frame | Base Frame |
| Msg Num | - | Msg Num | Msg Num |
| Server Time | Client Time | Server Time | Client Time |
| Sim Time | - | Sim Time | - |
| Num Devs | Ena Throttle | Name | Ena Pos Ctrl |
| Dyn Freq | Clock | Mass & Inertia | Pose |
| - | Jump Steps | Transform | Wrench |
| - | - | Children[] | Pos Ctrlr Mask[] |
| - | - | Joint Positions[] | Joint Cmds[] |

The necessity of using the dynamic time-step is that the simulation can run at any frequency while keeping the two clocks synchronized. Figure 8b depicts this behavior. It is important to mention the case where the dynamic-loop's frequency drops to $< 100Hz$, this mostly occurs during collision computation for a relatively high number of objects during contact or explicit throttling by learning agents using the **Ena Throttle**. In such cases, the time-step is still calculated dynamically, however, the number of sub-iterations the dynamic solver is allowed to progress at once needs extra attention. Bullet uses 3 parameters which include the time-step $\delta t$, maximum number of sub-iteration $N_{max}$ and the default integration time-step $\delta t_i$. The goal is to interpolate rather than recalculate the motion for $\delta t$ if:

$$\delta t < \delta t_i \times N \quad ; \quad N \in \mathbb{Z}^+ \quad \& \quad N \leq N_{max} \tag{8}$$

Ideally, $\delta t$ should not exceed $\delta t_i N_{max}$, but there is no guarantee this will not occur. Therefore, in order to have an accurate motion calculation for the degraded dynamic-loop's frequency, $N_{max}$ needs to be updated accordingly. However, increasing $N_{max}$ also increases the computational time which leads to circular deterioration. Finding the right balance between $N_{max}$ and $\delta t$ is challenging. However, to mitigate this limitation, $N_{max}$ is capped at a soft maximum.

Another proposed solution to avoid the degradation of the dynamic update-loop's frequency is to use collision primitives. Relatively advanced shapes can be created using a compound of various collision primitives. Primitive shapes have the advantage of utilizing implicit collision techniques rather than the costlier computation using state of the art algorithms such as Minkowski Distance and Gilbert Johnson Keerthi (GJK) algorithm [23]. Implicit collision computation is significantly faster and more reliable than explicit collision techniques, especially for a low-frequency dynamic update-loop. However, since creating collision primitives can be undesirable for complex shapes, mesh decimation techniques may be preferred.

## IV. Results and Discussions

A PC setup consisting of an Intel(R) Core(TM) i7-3770 CPU (3.40GHz), Fujitsu 32 GB DDR3 RAM (1333 MHz) and an Nvidia GTX 1060 (8 GB RAM) GPU running Ubuntu 18.04 was used for the demonstration of results. To test

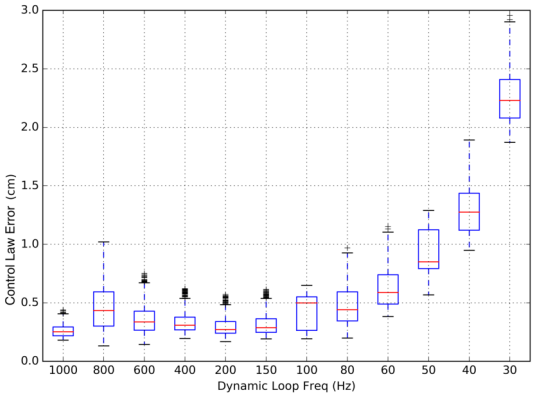



Fig. 9: Reponse of haptic controllers with degrading dynamic-loop's frequency

the Asynchronous Framework for Assistive Intelligence and Control, we analyzed the response of haptics and control of the Simulated Dynamic end-effectors (SDEs) as the dynamic simulation slows down. Since such a scenario is difficult to reproduce for performance metrics without careful preparation, we explicitly throttled the dynamic-loop's frequency by using the "step throttling" functionality discussed in Section III-F. Next, to generate consistent input motion profiles, we use mock devices by exploiting the **DVRK Arm** plugin interface (shown in Figure 6) to spawn two input devices (MTM-R and MTM-L) in the AMBF Simulator. We then generated a customizable trajectory of the form:

$$P_{input} = P_{off} + [a_p sin(t_c t), b_p cos(t_c t), c_p sin(t_c t)]' S \tag{9}$$

In the above equation $P_{input}$ is the commanded position of the Input Device while the R.H.S consists of offset $P_{off}$, time constant $t_c$, scale $S$, system time $t$ and $a_p, b_p, c_p$, which are the major/minor axes but in a 3 Dimensional space. In order to generate a high velocity, during our testing procedures, we set $tc = 4.0$, $S = 0.1m$ and $a_p = 1, b_p = 1, c_p = 2$. A script systematically throttles the dynamic-loop's frequency and records the controllers' performance as the magnitude of error from set-point. Figure 9 shows the output of the controllers performance for $n = 5000$ readings. It is evident that controllers' response only begins to suffer as the dynamic-loop's frequency falls below 60 $Hz$. The reason we are using the controller performance as such for haptic response is due to the shared-data structures discussed in Section III-A. For the force-feedback computation for both the device and SDEs, we only used the error from their set-points. The remaining control schemes, such as the gravity compensation, were added modularly from the device drivers or distributed controllers that were running in conjunction with the dynamic update-loop.

Next, we demonstrate an example for manipulating and solving a complex puzzle using two users controlling a pair of devices each (Figure 10). Two devices had haptic feedback (dVRK MTMs visible as PIP on the top right) while the remaining two were simply tracker devices with lower

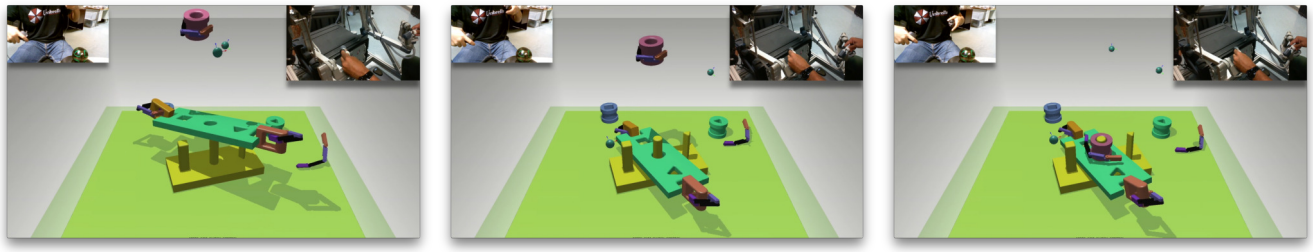


Fig. 10: These sub-figures show the progression (left to right) of a bi-manual task using the AMBF Simulator. The Two end-effectors holding the green multi-link puzzle piece are controlled by dVRK Masters (shown as Picture in Picture on top right) and the other two end-effectors are controlled via Razer Hydra (shown as Picture in Picture on top left)

| topic (name) | rate (Hz) | min_delta (s) | max_delta (s) | std_dev | window |
|---|---|---|---|---|---|
| /chai/env/Box_55/State | 1.967e+03 | 7.868e-06 | 0.01539 | 0.0006213 | 8516 |
| /chai/env/Box_56/State | 1.956e+03 | 8.106e-06 | 0.01444 | 0.0006179 | 8516 |
| /chai/env/Box_57/State | 1.962e+03 | 8.821e-06 | 0.01361 | 0.0005808 | 8480 |
| /chai/env/Box_58/State | 1.961e+03 | 7.868e-06 | 0.01471 | 0.000605 | 8480 |
| /chai/env/Box_59/State | 1.965e+03 | 6.914e-06 | 0.01263 | 0.0005788 | 8478 |
| /chai/env/Box_60/State | 1.954e+03 | 8.821e-06 | 0.01649 | 0.0005974 | 8478 |
| /chai/env/Box_61/State | 1.959e+03 | 9.06e-06 | 0.01357 | 0.0005925 | 8378 |
| /chai/env/Box_62/State | 1.954e+03 | 9.06e-06 | 0.01312 | 0.0005545 | 8378 |
| /chai/env/Box_63/State | 1.963e+03 | 7.868e-06 | 0.0136 | 0.0005484 | 8360 |
| /chai/env/Box_64/State | 1.958e+03 | 7.868e-06 | 0.01304 | 0.0005565 | 8360 |
| /chai/env/Box_65/State | 1.955e+03 | 7.868e-06 | 0.01275 | 0.0005779 | 8286 |

Fig. 11: Communication speed of several *afObjects* for an overloaded dynamic environment. The desired communication frequency is set to 2 kHz Dynamic-Loop's Frequency $\sim$ 300 Hz, *afObj*Comm frequency $\sim$ 2 kHz

update-rates (PIP on the top left). The puzzle involved several pieces including a multi-link Puzzle (Green Plate with Orange Handles), a Puzzle Base (Yellow Mesh) and three single link rigid body puzzles which included the Triangle Puzzle (Green cylindrical shape), Square Puzzle (Blue cylindrical shape) and the Circle Puzzle (Red cylindrical shape). The Puzzle base and the multi-link Puzzle had a matching set of extrusions and holes respectively, while the three rigid body puzzles had intruded cuts to match the three extrusions of the puzzle base. It is important to note that all of the grasping interactions in the simulation were purely dynamic. Hence, they involved a combination of friction due to contact geometry, grip force, slip, and slide. We did not use any simplification techniques for appending the grasped object as a fixed body to the end-effector link. While this dynamic grasping helps provide a natural feel by allowing gripping slack, it makes puzzle solving more challenging.

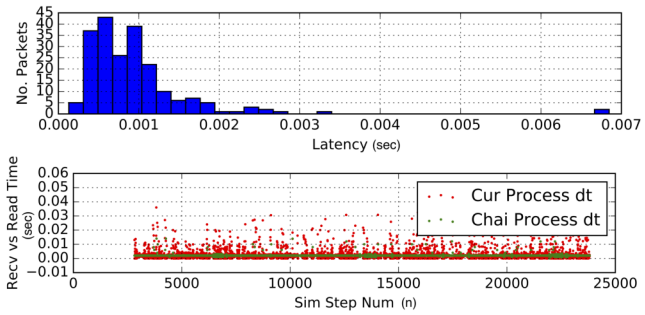



Fig. 12: (a) Histogram of the time difference between the embedded time of a received packet and the current time for synchronous communication using Step Throttling (b) Difference between embedded times of consequently received packets (green dots) vs the time they are read (red dots).

Figures 10 show the progression of a sub-task that involves manipulation of a multi-link puzzle. The multi-link puzzle requires at least two inputs to be lifted and placed on the Puzzle Base. This is followed by the Single link Puzzle pieces being placed on top. All of the four simulated end-effectors can interact with each other and the remaining puzzles. The close-loop constraint formed by the multi-link Puzzle is felt by the dVRK Masters which constrains the range of motion, thus allowing better control and manipulation.

Requirements that drive the design guidelines of the puzzle pieces are multi-manual manipulation, moderately complex maneuvering and ease of grasping. Given that the goal of this study was to demonstrate the flexibility and utility of the Asynchronous Framework, the puzzles utilized here were not developed formally for a particular clinial subtask. Moreover, designing these types of puzzles based on any specification is trivial using Solidworks / Blender or any mesh creation software. The corresponding lower-resolution collision meshes were generated using mesh-decimation techniques. The lower resolution collision meshes are helpful for maintaining higher frequency dynamic update-loop. In this study we used CPU for Physics computation and a GPU for graphics processing. The addition of Open-CL (Open Compute Language) for both rigid-body solvers and collision computation is a future goal.

For the purpose of training neural networks and agents for assistive intelligence, we discuss the interfaces exposed by the Python Client (Section III-E). The focus here is the ease of creating learning agents using tools such as GYM, Keras, TensorFlow/Theano and Keras-RL from the Python Client and not result of the trained models. The Python Client exposes compatible interfaces for Keras since each dynamic body in the simulation can be probed using *afState* and manipulated using *afCommand*. While it is trivial to use a limited number of dynamic objects (shown in Figure 10) for training NN or RL agents, we show the possibility of training a larger number of simulated bodies. In Figure 11, 200 dynamic boxes were added to the AMBF Simulator. ROS introspection tools were used to probe the frequency of *afState* for a few boxes with the desired communication frequency set to 2 kHz. Due to the excessive load on the AMBF Simulator, the dynamic-loop's frequency dropped to around 300 Hz. However, the communication speed for all *afObject*s was $\sim 2kHz$ as shown in Figure 11.

An essential feature of the Python Client is controlling the AMBF Simulator synchronously for satisfying Markov's action-reward pair property under the umbrella of the Asynchronous Framework. Immediate feedback is difficult to achieve in a distributed architecture since it involves delay due to (1) round-trip communication of packets and (2) processing time in between for computing kinematics, dynamics and updating packet data. The latency caused by the round-trip of data is presented in Figure 12(a) and (b). As illustrated in Figure 12 (b), the bottleneck is caused by the execution speed of Python. This jitter is expected as a result of the longer queue sizes. Hence, the latency increases as newer data needed to wait in a queue while the program execution processed older data. For synchronous control, however, we are concerned with the latest data, and thus, the queue-size is set to 1. The limited queue-size helps to drive down the round-trip communication latency to $\leq 0.001secs$ for 2 kHz of communication speed (Figure 12(a)).

With regard to the evolution of the Asynchronous Framework, we intend on integrating flexible body dynamics and visualizations to create realistic surgical training applications using simulated body tissues, organs, cloths and threads. The foreseeable challenges include the complexity of implementation, stability, performance, dynamic-update to track real-world clock and manipulation using haptic devices. Moreover, new guidelines for the communication interfaces *afState-afCommand* need to be developed. Lastly, the inclusion of flexible body dynamics should employ a generic framework design to allow for universal adaptation as opposed to a more targeted application.

In this manuscript, we discussed the motivation behind the design philosophy of an Asynchronous Framework for a distributed application that is intended for training learning agents via real-time input from a dynamic-haptic simulation. The entire framework is available at the public repository [24]. The challenges to such an implementation are discussed throughout the text, and we have concluded with the performance analysis of the proposed Asynchronous Framework.

## REFERENCES


[1] A. Shademan, R. Decker, J. Opfermann, S. Leonard, A. Krieger, and P. C. W. Kim, "Supervised autonomous robotic soft tissue surgery," Science Translational Medicine, vol. 8, pp. 337ra64–337ra64, 05 2016.

[2] K. Bumm, J. Wurm, J. Rachinger, T. Dannenmann, C. Bohr, R. Fahlbusch, H. Iro, and C. Nimsky, "An automated robotic approach with redundant navigation for minimal invasive extended transsphenoidal skull base surgery," Minimally invasive neurosurgery : MIN, vol. 48, pp. 159–64, 07 2005.

[3] S. Sen, A. Garg, D. V. Gealy, S. McKinley, Y. Jen, and K. Goldberg, "Automating multi-throw multilateral surgical suturing with a mechanical needle guide and sequential convex optimization," in Robotics and Automation (ICRA), 2016 IEEE International Conference on. IEEE, 2016, pp. 4178–4185.

[4] A. Murali, S. Sen, B. Kehoe, A. Garg, S. McFarland, S. Patil, W. D. Boyd, S. Lim, P. Abbeel, and K. Goldberg, "Learning by observation for surgical subtasks: Multilateral cutting of 3d viscoelastic and 2d orthotropic tissue phantoms," in Robotics and Automation (ICRA), 2015 IEEE International Conference on. IEEE, 2015, pp. 1202–1209.

[5] A. Krupa, J. Gangloff, M. de Mathelin, C. Doignon, G. Morel, L. Soler, J. Leroy, and J. Marescaux, "Autonomous retrieval and positioning of surgical instruments in robotized laparoscopic surgery using visual servoing and laser pointers," in Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on, vol. 4. IEEE, 2002, pp. 3769–3774.

[6] R. Bauernschmitt, E. U. Schirmbeck, A. Knoll, H. Mayer, I. Nagy, N. Wessel, S. Wildhirt, and R. Lange, "Towards robotic heart surgery: Introduction of autonomous procedures into an experimental surgical telemanipulator system," The International Journal of Medical Robotics and Computer Assisted Surgery, vol. 1, no. 3, pp. 74–79, 2005.

[7] K. Shamaei, Y. Che, A. Murali, S. Sen, S. Patil, K. Goldberg, and A. M. Okamura, "A paced shared-control teleoperated architecture for supervised automation of multilateral surgical tasks," in Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on. IEEE, 2015, pp. 1434–1439.

[8] T. D. Nagy and T. Haidegger, "An open-source framework for surgical subtask automation," Robotics and Automation (ICRA) Workshops, 2018 IEEE International Conference on, 2018.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," arXiv preprint arXiv:1509.02971, 2015.

[10] N. Heess, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. Eslami, M. Riedmiller, et al., "Emergence of locomotion behaviours in rich environments," arXiv preprint arXiv:1707.02286, 2017.

[11] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, "Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo," arXiv preprint arXiv:1608.05742, 2016.

[12] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[13] A. Deguet, R. Kumar, R. Taylor, and P. Kazanzides, "The cisst libraries for computer assisted intervention systems," in MICCAI Workshop on Systems and Arch. for Computer Assisted Interventions, Midas Journal, vol. 71, 2008.

[14] P. Kazanzides, Z. Chen, A. Deguet, G. S. Fischer, R. H. Taylor, and S. P. DiMaio, "An open-source research kit for the da vinci® surgical system," in 2014 IEEE International Conference on Robotics and Automation (ICRA). IEEE, may 2014.

[15] C. et al., "The CHAI libraries," in Proceedings of Eurohaptics 2003, Dublin, Ireland, 2003, pp. 496–500.

[16] E. Coumans, "Bullet physics simulation," in ACM SIGGRAPH 2015 Courses, ser. SIGGRAPH '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2776880.2792704

[17] C. et al., "Keras," https://github.com/keras-team/keras, 2015.

[18] M. Plappert, "Keras-rl," https://github.com/matthiasplappert/keras-rl, 2016.

[19] D. Shreiner and T. K. O. A. W. Group, OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1, 7th ed. Addison-Wesley Professional, 2009.

[20] A. Munawar and G. Fischer, "Towards a haptic feedback framework for multi-dof robotic laparoscopic surgery platforms," in Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on. IEEE, 2016, pp. 1113–1118.

[21] A. Munawar, "Plugin based Interface for the da Vinci Research Kit (dVRK) MTMs," https://github.com/WPI-AIM/dvrk_arm, 2016.

[22] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in ICRA workshop on open source software, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[23] M. Sagardia, T. Stouraitis, and J. L. e Silva, "A new fast and robust collision detection and force computation algorithm applied to the physics engine bullet: Method, integration, and evaluation," in Prof. of the Conf. and Exhibition of the European Association of Virtual and Augmented Reality (EuroVR), 2014, pp. 65–76.

[24] A. Munawar, "The Asynchronous Multi-Body Framework," https://github.com/WPI-AIM/ambf, 2019.