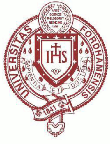

Fordham University

Fordham Research Commons

Faculty Publications

Robotics and Computer Vision Laboratory

Summer 2022

A Monte Carlo Framework for Incremental Improvement of Simulation Fidelity

Damian M. Lyons

James Finocchiaro

Misha Novitzky

Chris Korpela

Follow this and additional works at: https://research.library.fordham.edu/frcv_facultypubs

Part of the Robotics Commons

A Monte Carlo Framework for Incremental Improvement of Simulation Fidelity*

D.M. Lyons[1], J. Finocchiaro[2], M. Novitzky[2] and C. Korpela[2]

[1] Fordham University, Bronx, New York dlyons@fordham.edu
[2] United States Military Academy, West Point, New York
{james.finocchiaro, michael.novitzky, christopher.korpela}@westpoint.edu

Abstract. Robot software developed in simulation often does not behave as expected when deployed because the simulation does not sufficiently represent reality - this is sometimes called the 'reality gap' problem. We propose a novel algorithm to address the reality gap by injecting real-world experience into the simulation.
It is assumed that the robot program (control policy) is developed using simulation, but subsequently deployed on a real system, and that the program includes *a performance objective* monitor procedure with scalar output. The proposed approach collects simulation and real world observations and builds conditional probability functions. These are used to generate paired roll-outs to identify points of divergence in simulation and real behavior. From these, *state-space kernels* are generated that, when integrated with the original simulation, coerce the simulation into behaving more like observed reality.
Performance results are presented for a long-term deployment of an autonomous delivery vehicle example.

1 Introduction

Simulation tools are widely used in robot program development, whether the program/controller is built by hand or using machine learning. At the very least, simulation allows a robot programmer to eliminate obvious program flaws. The availability of physics engines [1] has produced simulations that can more accurately model physical behavior, making it more attractive to use simulation in conjunction with machine learning techniques [2] [3] to develop robot programs. However, a robot program validated with simulation, when operating in a real, unstructured environment may come across phenomena that its designers just did not know to include in the simulation, even though the phenomenon could in fact be simulated if it were known a-priori to be relevant. Examples of this kind of simulation *'reality gap'* include inaccurate robot joint parameters, surface friction, object masses, sizes and locations.

* The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Department of Defense or the U.S. Government. Lyons was partially supported by DL-47359-15016 from Bloomberg LP.

This paper addresses closing the *reality gap* for simulations used to develop robot programs by combining *sim-to-real* [4] and *real-to-sim* [5] approaches. We will assume that the simulation is a black-box and we present a framework to coerce simulation behavior to more closely resemble real experience.

The simulation has configuration parameters ϕ. If deployment of the robot program (π) results in failure, as determined by a performance monitor, then our overall objective is for ϕ to be updated by the deployment experience and π redeveloped to handle that experience. We expect this virtuous cycle of simulation, deployment and improvement to iterate. The simulation will be modeled as a transition function $T_\phi(s'|s, a)$ where s is the current sensor data from the simulation, a is the action to be carried out, s' is the resulting sensor data after the action is taken, and ϕ is a setting of the configuration parameters for the simulation. Real world experience will also be modeled as a transition function $T_r(s'|s, a)$. Two important novel aspects of our work are 1) a domain independent proposal for ϕ (as opposed to the more domain specific examples of domain randomization, e.g., [6] [7]) and 2) a Monte Carlo method to modify ϕ based on a direct comparison of estimated transitions functions (as opposed to the comparison of observations).

Fig. 1: Modified Traaxis platform used in field trials (left); simulation model (right).

The remainder of the paper is laid out as follows. Section II is a review of related work. Section III presents our method and architecture. Section IV details an introductory, simulation-only example. Section V describes the field trial, using simulation for design and a customized Traaxis platform for deployment, Fig. 1. Section VI presents our results in two parts. The first part addresses the unexpected environment change observed in deployment. The second part addresses unexpected sensor performance. We show that data collected during deployment can be used to make the simulation behave more realistically, iteratively narrowing the reality gap. Conclusions and next steps are presented in Section VI.

2 Related Work

Researchers using learning techniques to train robot programs on simulation have developed several approaches to the simulation reality gap. The *sim-to-real* approach addresses the issue of moving a policy trained in simulation to real hardware; [4] discusses this new direction and its open challenges, one key one

of which is improving the accuracy of the simulation tools used - where we aim to contribute.

Estimating a model of an observed system is typically a system identification [8] problem. The problem addressed here differs in two ways: Firstly, the system addressed is not the robot to be controlled, but rather the environment in which the robot will be embedded. Secondly, the model to be constructed is an 'add-on' model for an existing 3D simulation that employs its own blackbox model of the environment to overrides the simulation in certain parts of the state space and make its performance more accurate. In this sense, the work is similar to the *real-to-sim* approach, a complementary approach [5] in which real-world information is transferred to a simulation. We employ a composite local model approach, e.g., [9], to address the integration of reality and simulation.

Producing more realistic behavior does not just concern an improved physics engine and rendering technology, but also concerns handling uncertainty about which specific environment the robot will encounter. Benjamin [10] and Lyons [11] proposed an approach for cognitive robotics where a simulation is used to predict the environment state and to visually compare video with simulation predicted imagery. Differences are used to modify the simulation, changing object appearance, positions and velocities to match the observed video. While this approach has similarities to mapping (e.g., SLAM), a key difference is its attempt to insert object descriptions from camera information into a 3D simulation so as to be able to leverage the predictive properties of the simulation. The approach relied on a graybox view of the simulation and its modeling of objects.

Christian et al. [12] use a simulation to learn policies for a number of tasks and consider transferring the policy from simulation to real robot. They note that the simulation policy is generally correct in high-level gist but fails on some lower-level details. Yu [13] proposes a two stage approach to policy training for bipedal locomotion: a presimulation step to ballpark the system identification for simulation, followed by a more accurate tuning at deployment. Permana [14] trains a CNN on synthetic imagery for visual detection of ground casualties, and handles the sim-to-real issue by injecting noise, downsampling, segment removal and other changes to the simulation data.

Adopting the domain randomization approach, Peng et al. [2] use random modification of 95 simulation parameters to show that even low fidelity simulation can be used to train a robot to push a puck successfully over a wide range of real-world situations. Cheboter et al [15] interleave many simulation runs with much fewer real-world runs, and modify the simulation configuration so that it simulates a range of situations more similar to observed experience. Our proposed approach is most like that of Chebotar. However, instead of using the domain randomization — establishing a range of initial parameter values — we follow an approach more similar to Benjamin and Lyons [11] who modify simulation state during a simulation to close the reality gap. Our approach differs from sim-to-real approaches in putting the emphasis on improving the simulation via real-to-sim so that the sim-to-real step becomes less onerous. It differs from [11]in adopting a blackbox view of the simulation.

3 Approach

The simulated and physical environments are modeled as Markov Decision Processes (MDP) $M_{sim} = (S, A, T_{sim}, R)$ and $M_{phy} = (S, A, T_{phy}, R)$ that differ only in their transition function. The robot program implements a policy π that can be applied to either MDP. The sensors available to the robot program are $SN = sn_0, sn_1, \ldots, Sn_n$ with value sets $SV = Sv_0, Sv_1, \ldots, Sv_n$. The control outputs available are $AN = an_0, an_1, \ldots, an_m$ with value sets $AV = Av_0, Av_1, \ldots, Av_m$. The actions available to the robot are any setting of the control outputs

$$A = Av_0 \times Av_1 \times \ \ldots Av_m \tag{1}$$

and the set of states S is

$$S = Sv_0 \times Sv_1 \times \ \ldots Sv_n \tag{2}$$

A reward function, R, will be used as a measure of when the robot program is achieving its performance objectives on M_{sim} or M_{phy}. The software to measure R can either be hand-written by a software engineer, generated automatically from task specifications [16], or, if the program was generated by learning, copied from the learning reward module.

The MDP transition functions are defined

$$T_{sim}, T_{phy} : S \times S \times A \to [0, 1] \tag{3}$$

and are interpreted in their usual way as the conditional probability $T(s'|s, a)$ of transitioning from state s to state s' when action a is carried out. These functions encapsulate any difference between what occurs when an action a is carried out in the simulation in a state s and when the same action a is carried out in state s in the real environment.

3.1 Kernel Generation

Once the program is ready to be deployed, we collect data from multiple runs of the final version of the robot program through the simulation:

$$H_{sim} = \{(s_i, a_i, r_i, s'_i) : 0 \leq i \leq I_{max}\} \tag{4}$$

for all action a taken in state s then resulting in state s' and reward r in a run of the simulation. Similarly, H_{phy} is collected for the final program in deployment – representing real experience. Using a frequentist approach, T_{sim} is estimated from H_{sim}, and T_{phy} from H_{phy}. The state to state transition function P that is the combination of the T (for each function) with π, the control software is:

$$\begin{aligned} B(s'|s, a) &= T(s'|s, a)\pi(a|s), \ a \in A, s, s' \in S \\ P(s'|s) &= \textstyle\sum_{a \in A} B(s'|s, a) \end{aligned} \tag{5}$$

Let $SR(s)$ be the set of state, action, probability tuples that directly transition from state s to state s' in P:

$$SR(s) = \{(s', a, B(s'|s,a)) : B(s'|s,a) > 0\} \tag{6}$$

Since a state only includes what is observable to the robot, there may be hidden dynamics in the real world that result in $SR_{phy}(s)$ differing from $SR_{sim}(s)$. Our first objective is to identify this difference. We define the policy roll-out from a state s as a sequence:

$$Roll(s) = \begin{cases} SR(s) \neq \emptyset & (s', a, p).Roll(s') \ (s', a, p) \in SR(s) \\ else & \perp \end{cases} \tag{7}$$

Where '.' is sequence concatenation and $\perp$ indicates end of sequence. A number of *paired roll-outs* are calculated $Roll_{sim}(s)$ and $Roll_{phy}(s)$ and compared up to the point at which they are considered to diverge. The distance between two states is defined by the weighted L1 norm $c(s, s')$:

$$c(s, s') = \sum_i w_i |s_i - s'_i|, \ \sum_i w_i = 1 \ w_i \in \Re, \tag{8}$$

States differ if their distance is greater than a threshold ϵ_c. If states differ, and they have sufficiently different probabilities p under each transition function, this is called *a roll-out divergence*. For each divergence detected, we construct *a state-space kernel* that we will use to modify the behavior of the simulation in that region of its state space so that it is more similar to observed behavior of the physical environment.

Next we will describe how the kernel is built, then how the simulation is modified by a kernel. The key pieces of information in a kernel k are:

- The *region of state space* in which it is active: $e_k = N(0, \sigma^2)$ is a univariate normal distribution of distance from s, the state preceding the divergence, where $c_k(.) = c(s, .)$ is used to calculate the scalar distance between s and any other state. For any state s', $e_k \circ c_k(s')$ is measure of how active kernel k is at s'.
- The *divergence probability distribution*: $\wp_k = (p_s, p_p)$, where $\delta = (s, a, p)$ is the common roll-out point just before divergence and $\delta_s = (s_s, p_s, a)$ in $Roll_{sim}(s_0)$ and $\delta_p = (s_p, p_p, a)$ in $Roll_{phy}(s_0)$ are the points after divergence.
- The *kernel transfer function*: $f_k : S \times A \to S$. This mapping is a composition of linear functions f_{k,sn_i} for each sensor $sn_i \in SN$ constructed by least-squares fit to the state and action data in the sequence starting with the state preceding divergence and including a small number of successor states. We will restrict the kernel to be active in a small region of the state space, and since the transfer function estimate is in a small region of the state space, we argue that this can serve as a piecewise linear approximation of a more complex function. For now, we omit functions that transform a state back to itself. We discuss this useful special case later in the paper.

Algorithm 1 shows the algorithm for kernel construction.

Algorithm 1 Generate Kernels

```
procedure GenKer(P_sim, P_phy)
    S_0 = States(P_sim) ∩ States(P_phy)
    while i < N_rollouts do
        s_0 ~ U(S_0) // sample uniform distrib
        r_s, r_p = Roll_sim(s_0), Roll_phy(s_0)
        δ, δ_s, δ_p = divergence(r_s, r_p) // state δ followed by δ_s in r_s & δ_p in r_p.
        if δ ≠ null then // construct kernel at δ
            e = N(s, σ^2) // δ = (s, a, p) eq.(6)
            d = normalized(p_s, p_p) // where δ_s = (s_s, a_s, p_s), sim. for δ_p
            f = linearfit(δ, δ_p, ...) // arg. is a short subseq of r_p
            AddKernel(e, d, f)
        end if
    end while
end procedure
```

3.2 Kernel Manager

We consider the simulation as a black box process, but one over which we have some control. For example, in [15] a black-box simulation is configured by selecting initial parameters ϕ from a distribution P_ϕ. They attempt to learn the distribution P_ϕ that best matches traces of real stored experience. We will also assume that we have access to a set of parameters ϕ for the black box simulation but more similar to those of [11] which effect the simulation *on simulation each time-step* and not just initialization.

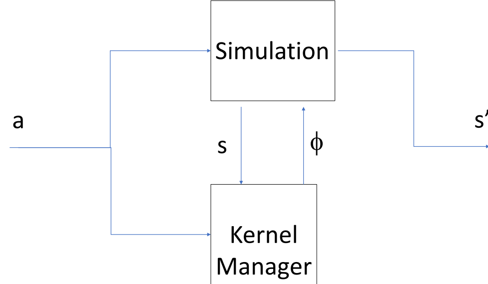

Fig. 2: Architecture for kernel modified simulation

At each simulation time step, the robot program sends its selected action a to the simulation. While the internal state of the simulation is unknown, it generates the sensor output values, the M_{sim} state s, and this is communicated to the *Kernel Manager*. The kernel manager checks first to see whether the state s is within the spatial scope for any kernel k by evaluating $e_k \circ c_k(s) > \epsilon_c$ (for small ϵ_c) to see if that kernel is active. The kernel manager selects the first active kernel, if any are active, and uses it to modify the state of the simulation, using the simulation parameters ϕ so that the simulation returns $f_k(s, a)$. The kernel

manager itself has no knowledge of the internal simulation state and dynamics. Knowledge of the simulation is encapsulated in the design of the simulation configuration parameters ϕ so that values of ϕ will modify simulation internal state to ensure that $f_k(s, a)$ is generated.

Although a kernel is only active in a small region of the state space, the effect of any change to the simulation state is potentially long lasting. This results in both a spatial and temporal generalization effect. A modification of an object location for example, will result in sensors picking up the revised location no matter the pose of the robot - a spatial generalization. Other simulation entities that subsequently interact with the object (e.g., collide) will do so at the new location - a temporal generalization. Both of these generalizations leverage the simulation dynamics applied to new experience gained from execution of the robot program in a real environment.

4 Delivery Vehicle

We will consider an autonomous delivery vehicle as a long-term service deployment example. The vehicle will acquire goods from a storage location and deliver to two outlying sites, returning again to the start location. For this paper, we will simplify the task to just the transit portion of the problem. The vehicle repeatedly makes the trip from stores to each delivery location and then back to stores in a long-term outdoor deployment; each trip is one mission. The performance measure for the vehicle is that it reach each goal g_i within a specified position accuracy ϵ_i and time limit d_i:

$$perf(t) = \begin{cases} 1 & |p(t) - g_i| < \epsilon_i \;\&\; d(t) \leq d_i \\ 0 & |p(t) - g_i| > \epsilon_i \;\&\; d(t) \leq d_i \\ -1 & d(t) > d_i \end{cases}$$

where $p(t), d(t)$ are the position of the robot and time since last goal, respectively.

During the design phase, the robot program is tested in simulation using what is known about the problem a-priori. The deployment vehicle is a modified Traaxis platform with Ackermann steering (described in more detail later) operating in large open field area. Fig. 3(a) and 3(b) show the intended deployment platform and a Gazebo simulation of the platform (using the open source Ackermann_vehicle model[3]). Fig. 3(c) shows the Gazebo simulation of the open field area with the three locations (store and two delivery sites) shown by lighter colored patches.

The task is programmed in Python using ROS (Robot Operating System). The main loop sends the the robot to each of the waypoints in turn and initializes the performance monitor with the goal location, time deadline and accuracy. A saturated spring model is used to calculate speed and a bicycle pursuit model used to generate Ackermann steering. These are linearly transformed to Traaxis

[3] github.com/trainman419/ackermann_vehicle-1

motor commands. The position of the robot at the start of each mission is selected from a uniform distribution and a small amount of zero mean noise is injected into the speed and steering signals. GPS data was simulated by providing Gazebo model position information at a rate of 20 Hz and vehicle orientation was calculated from GPS velocity. Vehicle pitch sensing was also simulated from model information.

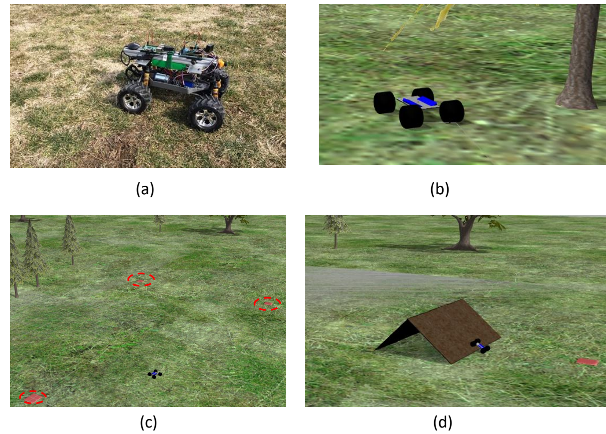

Fig. 3: Real robot (a), simulated robot (b); simulated delivery terrain with waypoints highlighted (c), unexpected terrain (d)

As discussed, the inital robot program could be handwritten or developed using a method such as reinforcement learning or any combination of these. All we insist is that the performance measure is in place and the robot program (once ready for deployment) is validated in the simulation. The performance measure is used to assign a reward and for the purposes of this paper the two phrases 'reward' and 'performance measure' can be considered synonymous.

The Average Total Reward (ATR), the sum of all rewards divided by number of missions, is used as a measure of success. Fig. 4 shows the ATR graphs for the design and deployment of this example. The graph labeled "Design" is the ATR from validating the robot program in the simulation. At this point in the design process, the software designers have all the information they consider necessary to have the program fulfill their understanding of the specification. Thus, we consider the ATR graph to be an objective characterization of success.

During a long-term deployment, the environment may exhibit phenomena not anticipated by software designers. For this experiment, we added a ramp between two of the three waypoints (Fig. 3(d)). The vehicle can navigate the ramp, but it does slow it down and will almost consistently cause it to fail to meet one of its deadlines. The ATR graph in Fig. 4, labeled "Deploy" shows the performance of the control software in this modified simulation (which is taking the place of a real deployment for our example purposes). It is lower than the

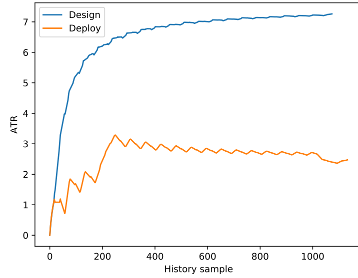

Fig. 4: ATR graphs for robot program at design and deployment for example

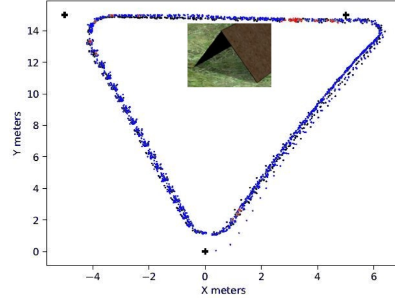

Fig. 5: Position scatter graph, H_{sim} (black), H_{phy} (blue); calculated kernel posn (red). '+' marks goal locations. 'Unexpected' ramp location shown.

design ATR. We consider this as an empirical indication that the robot program is not meeting its design specification; however, we can't say yet what is different about the environment at deployment time that caused this to happen.

The robot program is instructed to record H (eq.(4)). The state in this example includes the robot odometry and pitch. It is important that all sensing information is recorded, not just that (currently) used in the task. The action value is the robot speed and steering topics. The reward is the value from the performance metric. H_{sim} and H_{phy} were collected for the missions graphed in Fig. 4. The kernel generation algorithm was run on this data and 38 unique kernels were constructed (omitting duplicates and candidate divergences that did not have sufficient information to estimate the transfer function).

The robot program is rerun through the simulation receiving configuration commands from the Kernel Manager (Fig. 2), generating the ATR graph labeled "Kernels" in Fig. 6. It is similar to the "Deploy" graph for an initial period but overall it is not as poorly performing as real deployment while still much worse than the "Design" curve. The simulation, modified by the deployment experience, now much more closely represents what happens when the delivery robot operates in its physical environment. This provides a tool for designers to redesign the program so that it will work better in the physical environment.

The robot program could be redesigned manually, using the kernel-modified simulation for testing. For this example, we chose to use a SARSA reinforcement learning algorithm in conjunction with the kernel-modified simulation to select modified control values that maximized the reward. The resulting robot program issued faster velocity values when non-zero pitch values were encountered. The redesigned program, the "Redesign/K" graph in Fig. 6, shows substantially improved performance with respect to the "Kernels" graph.

Finally, the redesigned robot program was run again in the "Deploy" situation - that is, in the simulation that included the unexpected ramp, and without any Kernel manager input. The graph in Fig. 6 labeled "Redeploy" shows the ATR for this case. It is higher than the original "Deploy" graph and close, but not identical to, to the original "Design" graph. The simulation, enhanced by the kernels automatically extracted from prior physical experience, is now a bet-

ter reflection of the physical environment and can be used to produce a robot program that will work better in the physical environment.

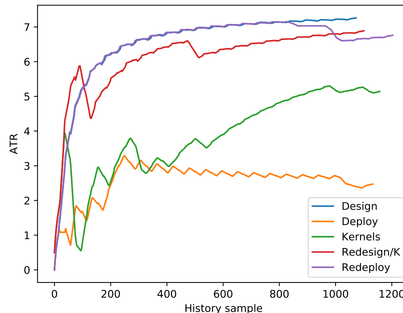

Fig. 6: ATR graphs for robot program redesign

5 Field Test

To evaluate the proposed approach, we prototyped a robot program for a three GPS waypoint mission using Gazebo and ROS. The robot program was then deployed to a physical robot for traversing the GPS waypoints in an open field, with the unexpected situation that the robot sometimes encountered a steep ramp between two of the waypoints. The mission differed from that of the previous section in that the deadline and accuracy for performance related to the entire mission, from first to last waypoint, instead of per waypoint, and of course in that the deployment was not a simulation. The short deployment trial experience was used to generate state-space kernels, which were used to reduce the reality gap for the simulation. The experimental procedure and mechanism is described in more detail below, and the results are presented in the next section.

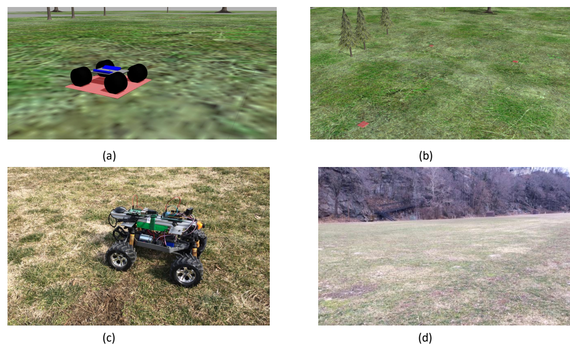

Fig. 7: Simulation robot (a) and course (b), and real robot (c) and course (d)

5.1 Design and Simulation

Design and simulation work was completed using ROS Indigo on a Dell Latitude 3460 laptop. The mission program was written in Python 2.7 and simulated using Gazebo with the open source Ackermann_vehicle model as before, Fig.

7(a), and with the same waypoint coordinates, Fig. 7(b). The robot program used a saturated spring model to generate velocities towards waypoints, and a bicycle pursuit model to generate the steering and speed for the vehicle, running at 10 Hz. However, preliminary experimentation with the real platform showed that the GPS data could certainly not be provided at 20 Hz, as it was in the previous section. Instead, GPS data was simulated by providing Gazebo model position information at a rate of 1Hz, and vehicle orientation was calculated from GPS velocity. Vehicle pitch sensing was calculated as before.

To compensate for the slow GPS, Ackermann kinematics were used to conservatively interpolate position between GPS samples. A small amount of uniformly distributed noise was added to the steering and speed signals to simulate moving on bumpy grass. The performance requirement was that the vehicle complete the course within a time deadline of 39 seconds (empirically established for the experiment) and with an accuracy of 2m of the final waypoint. The robot program was instrumented to collect all sensor, control and reward signals at 0.5 Hz.

5.2 Description of the Field Trials

The field trials were performed at the River Courts located at the United States Military Academy (USMA) in West Point, NY. The Robotics Research Center (RRC) at USMA uses the River Courts to perform robotics testing for aerial and ground robots. To facilitate testing, they have a trailer designed to be used/moved to remote locations for field trials and provide researchers with electricity, wifi, and climate control.

A significantly modified Traaxas Stampede 4x4 VXL, a four-wheel drive remote controlled vehicle, was used as the base of our platform for conducting experiments. Attached to the constructed frame was a Raspberry Pi 3 B+ and an Arduino Mega. The Arduino Mega served as a ROS node for the Raspberry Pi and returned IMU data from an Adafruit LSM9DS1 sensor [17].

The ROS mission program developed in simulation was also used for the field trials, with the exception that GPS location and vehicle pitch were read from topics published by the Arduino node. The relationship between the PWM control signals and the mission program steering and speed signals was established empirically and very coarsely. The lack of careful calibration between simulation and physical platform was purposeful, since closing that gap is part of the objective of the research reported here.

6 Results

The data collected from the design and deployment phase is shown in Fig. 8. The scatter plot in Fig. 8(a) shows all the logged position samples of the simulated robot during the design phase of the work. The track differs from that in Fig. 5 because of the slower GPS rate: While the straight line segments are similar, the turns at each waypoint show some overshoot.The scatter graph of robot

locations of the physical robot during deployment is shown in Fig. 8(b), and it differs from both of the prior simulation graphs.

The Average Total Reward (ATR) graph shows that the simulation performs well according to its performance monitor. The deployed robot program fails to meet the performance requirement however, as evidenced by its decreasing ATR graph. Of course, we engineered the field trial to force this effect by introducing unexpected terrain, a physical ramp between waypoints (5,15) and (-5,15), Fig. 3(d).

As described in Algorithm 1, the data collected from design and deployment (Eq.(4)) was used to generate the transition functions P_{sim} and P_{phy} (Eq.(6)). P_{sim} had 951 state transitions, and P_{phy} had 942 transitions. The vast majority of both had a branching factor of 1. There were 882 states in common between the two, where states are compared as described in Eq.(8) and its accompanying text. One hundred paired roll-outs were conducted and divergent states identified. From this, 15 unique divergences were identified and kernels developed for them. Each of the kernels included a linear transfer function mapping velocity and sensor state to a new sensor state - position, orientation and pitch. Any kernel whose transfer function mapped each state to itself was omitted - for now. These will be discussed in a subsequent section.

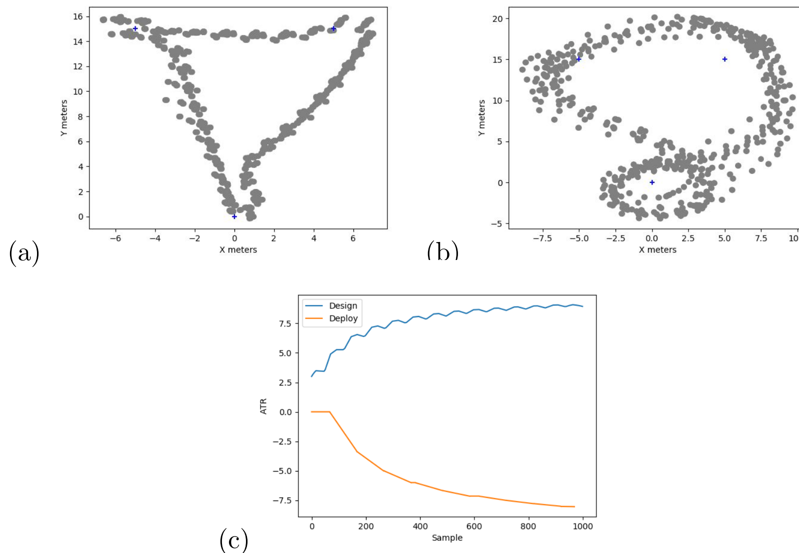

Fig. 8: Data from design and deployment trials (a) scatter plot of 2D position in design, (b) in deployment, and (c) combined ATR graph. Waypoint locations are marked as '+'.

6.1 Kernels non-identity transfer functions

Fig. 9(a) shows the kernel mean location overlayed on the deployment tracks. Recall that this location is the one immediately preceding the divergence - and on

the track overlay, many kernels are clustered around and just after the waypoint preceding the ramp.

The robot program was then rerun in the simulation but with the Kernel Manager active. Fig. 9(b) shows a number of ATR graphs including the original design and deployment graphs. The graph labeled "Kernels" shows the result of executing the original robot program but in the kernel modified simulation. The performance is much worse than the original performance, but not as severe as the actual deployment.

There are many ways that a simulation could be coerced into failing the performance measure. But unless the mechanism failure allows a designer or learning algorithm to redesign and test in simulation and have this generalizes to improved real performance, the method would have limited use. In section 4, the modified simulation was used in conjunction with a SARSA reinforcement learning algorithm to generate an improved robot program that generated faster velocities when non zero pitch was detected. That same same solution was applied here, but the velocity was manually adjusted so that the performance measure was achieved in the modified simulation. Our approach is agnostic as to the specific method of robot program improvement (machine learning in section 4 and manual modification here).

This improved robot program was tested on the kernel modified simulation, producing the ATR graph labelled "Redes/K" in Fig. 9(b). The modified robot program shows improved performance over the original program, approaching within ~25% of the original performance. The modified program was redeployed to the robot, and that performance is shown in the ATR graph labeled "Redeploy" in Fig. 9(b). The improvement in behavior transfers from simulation to reality, supporting the argument that robot programs (implementations of control policies) developed with the kernel mechanism transfer well to reality.

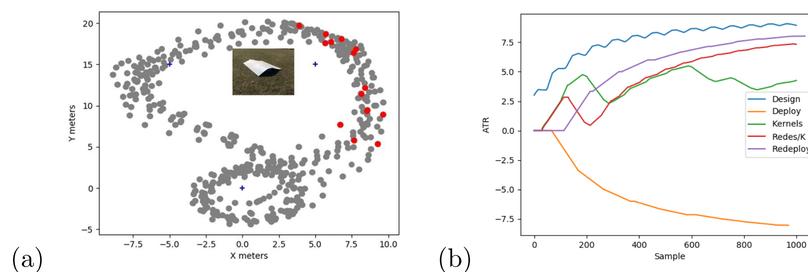

Fig. 9: Deployment track data (a) overlay showing calculated kernel locations in red, where ramp is shown at its approx. deployed location, (b) ATR graph for kernels, redesign and redeployment.

6.2 Kernels with identity transfer functions

Our inital analysis rejected any kernels with identity transfer functions. This allowed us to identify the behavior divergences in roll-outs related to the intro-

duction of the ramp. However, these divergences don't address the reason that Fig. 8(a) is quite different in appearance from Fig. 8(b). To capture this, we need to discuss kernels that have an identity transfer function - these represent divergences in which the position sensor did not report any update of the position in the deployment roll-out.

This phenomenon is observed because the deployment GPS rate is lower than was expected at design time. However, the predictive model used at design time is unable to compensate, so that the position invariably lags, resulting in extensive overshoots. Unfortunately, our existing approach will not work here. If a kernel predicts no motion of the platform, then in addition to the sensors being caused to report no motion, the simulation is also modified to force no motion. The only way to distinguish a kernel from a sensor divergence, which we are discussing now, from that from a motor divergence, which is what we have addressed up until now, is to utilize a sensor timestamp, extending state as

$$S = Stv_0, \times Stv_1 \times \ \ldots Stv_n \qquad Stv_i = Sv_i \times T \tag{9}$$

For a set of times $T = \{t_0, t_1, \ldots\}$. Each roll-out will thus include timestamped sensor data. If f_{k,sn_i} is the transfer function for active kernel k sensor sn_i, and the kernel manager applies this to the current state $s = (stv_0, stv_1, \ldots, stv_n)$, then the modified state $s' = f_k(s,a)$ has $stv'_i = f_{k,sn_i}(stv_i, a)$. Finally, since $stv_i = (sv_i, t_i)$, if f_{k,sn_i} is an identity transfer function, then $t_i = t'_i$ and this is unambiguously from a sensor divergence. Any other transfer function is from a motor divergence. For convenience, these will be referred to as sensor kernels and motor kernels respectively.

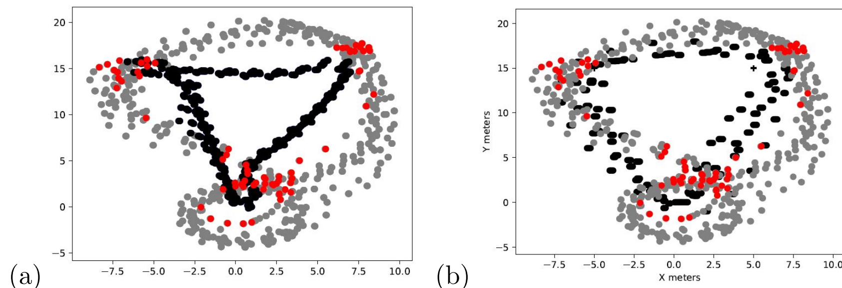

Fig. 10: Sensory Kernels: (a) overlay showing design track (black), deploy track (gray) and calculated kernel locations (red), (b) overlay of kernel modified design track (black) and original deploy track (gray).

The kernel manager treats these cases slightly differently: an active sensor kernel modifies the sensor data only, whereas an active motor kernel modifies sensor data and simulation state. When sensory divergences are *now not removed* from the one hundred paired roll-outs, 54 sensory kernels are identified in addition to the motor kernels already identified. Where the 15 motor kernels relate principally to the unexpected ramp, the 54 sensory kernels relate principally to the slower than expected GPS. Fig. 10(a) shows the location of the sensory kernels. They are focused around the turns at the waypoints. Fig. 10(b) shows an

overlay of the original deployment track (gray) with the new, kernel modified simulation tracks. The mission program was not changed to produce this track, which is now more similar to the original deployment; the change results from the effect of the sensory kernels in Fig. 10(a) extracted using our approach.

7 Conclusions

This paper has proposed a novel Monte-Carlo approach to the reality gap problem. The approach collects simulation and real world observations and builds conditional probability functions for them. These are used to generate paired roll-outs and to look for points of divergence in behavior. The divergences are employed to generate state-space kernels coercing the simulation into behaving more like observed reality within a region of the state space. Our results support not just that the kernel approach can force the simulation to behave more like reality, but that the modification is such that a robot program with an improved control policy tested in the modified simulation also performs better in the real world.

The kernel managed simulation in our field trials did not produce as poor an ATR graph as the actual deployment Fig. 9. This is principally due to restricting the scope of each state-space kernel to a small region of state space with $e_k \circ c_k(s') < \epsilon_c$. The advantage is that it allows us to leverage a fast function approximation method. The disadvantage is that it could require making a lot of kernels to capture the real environment with high fidelity. We argue however that there is an advantage in understating the effect of experience gained in each iterative deployment since it produces a sequence of smaller learning obstacles rather than a single large obstacle. This is a useful learning strategy [18].

The work most similar to ours is that of [15]. A crucial point of difference is that [15] (and some others) address the reality gap problem with a principled domain randomization approach, providing a range of environment for policy development, and generating a policy that is robust along the right dimensions of variability. We address the same problem but from the perspective of making each simulation run more closely resemble reality. This is reflected in how the simulation is "wrapped" by each approach: our approach requires a more invasive configuration – access to the Gazebo model information – but we argue that our configuration is less application specific and more easily generalized. We are especially interested in generalization to higher-dimensional state-spaces.

The quality of the improvement that our method makes is proportional to the divergence information that it has to work with. An avenue of future study is development of control strategies that improve divergence information by exploring more of the state space during deployment.

References

1. M. Reckhous, N. Hochgeschwender, J. Paulus, S. Shakhimardanov, and G. Kraetzschmar, "An overview about simulation and emulation in robotics." *Intl. Conf. on Sim, Modeling & Prog. for Aut. Rob.* pp. 365–374, 2010.

2. X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, “Sim-to-real transfer of robotic control with dynamics randomization,” *IEEE Int. Conf. Rob. & Aut.* 2018.
3. R. Moeckel, Y. Perov, A. Nguyen, M. Vespignani, S. Bonardi, S. Pouya, A. Sproewitz, J. van den Kieboom, F. Wilhelm, and A. Ijspeert, “Gait optimization for roombots modular robots - matching simulation and reality,” *IEEE/RSJ Int. Conf. Int. Rob. & Sys.* 2013.
4. S. e. a. Hofer, “Perspectives on sim2real transfer for robotics: A summary of the RSS 2020 workshop,” *arXiv:2012.03806 [cs.RO]* 2020.
5. Y. Takayama, P. Ratsamee, and T. Mashita, “Reduced simulation: Real-to-sim approach toward collision detection in narrowly confined environments,” *Robotics*, vol. 10, no. 131, 2021.
6. J. M. Tobin, W. Zaremba, and P. Abbeel, “Domain randomization and generative models for robotic grasping,” *IEEE/RSJ Int. Conf. Int. Rob. & Sys.* 2017.
7. A. Dehban, J. Borrego, R. Figueiredo, P. Moreno, A. Bernardino, and J. Santos-Victor, “The impact of domain randomization on object detection: A case study on parametric shapes and synthetic textures,” *IEEE/RSJ Int. Conf. Int. Rob. & Sys.* pp. 2593–2600, 2019.
8. L. Ljung, “Perspectives on system identification,” *17th IFAC World Congress*, vol. 41, no. 2, 2008.
9. V. Verdult, L. Ljung, and M. Verhaegen, “Identification of composite local linear state-space models using a projected gradient search,” *Int. J. Cont.* (75)16-17 2001.
10. P. Benjamin, D. Lyons, and C. Funk, “A cognitive approach to vision for a mobile robot,” *SPIE Multisensor, Multisource Inf. Fusion* 2013.
11. D. Lyons and P. Nirmal, “Navigation of uncertain terrain by fusion of information from real and synthetic imagery.” *SPIE Multisensor, Multisource Inf. Fusion* 2012.
12. P. Christiano, Z. Shah, I. Mordatch, J. Schneider, T. Blackwell, J. Tobin, P. Abbeel, and W. Zaremba, “Transfer from simulation to real world through learning deep inverse dynamics model,” *arXiv 1610.03518*, 2016.
13. W. Yu, V. C. Kumar, G. Turk, and C. K. Liu, “Sim-to-real transfer for biped locomotion,” *IEEE/RSJ Int. Conf. on Int. Rob. & Sys.* 2019.
14. R. P. Saputra, N. Rakicevic, and P. Kormushev1, “Sim-to-real learning for casualty detection from ground projected point cloud data,” *IEEE/RSJ Int. Conf. Int. Rob. & Sys.* 2019.
15. Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox, “Closing the sim-to-real loop: Adapting simulation randomization with real world experience,” *IEEE Int. Conf. on Rob. & Aut.* May 2019.
16. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, “Introduction to runtime verification.” *Lectures on Runtime Verification LNCS Volume 10457*, 2018.
17. “Adafruit lsm9ds1 imu,” https://www.adafruit.com/product/3387, 2020-02-27.
18. R. Wang, J. Lehman, J. Clune, and K. O. Stanley, “Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions,” *ArXiv*, vol. abs/1901.01753, 2019.