# Evolutionary Robotics: Crossing the reality gap with a little help from your GPU and domain randomization

Zedan Hussain
*Department of Informatics*
*University of Oslo*
Oslo, Norway
zedanh@uio.no

Vetle H. Olavesen
*Department of Informatics*
*University of Oslo*
Oslo, Norway
vetlehol@uio.no

***Abstract*—Computer simulation for training robotic controllers has become popular in recent times, and training through simulation is more practical than real life in many ways. Tasks such as resetting the robot and supervision of training is difficult and time consuming in real life, as well as it might inflict damage on the physical robot which leads to further resources having to be spent. Training in simulation comes with its own unique problems, mainly the reality gap which is the difference between simulation- and real life-performance. If we can mitigate this reality gap we can reduce training times for robot controllers, as well as reduce the cost of creating these models. In this paper we employ an ant-model with a fairly simple controller optimized by a genetic algorithm in an attempt to minimize the reality gap by randomizing the environment during our simulations, such as changing gravity and friction. Employing domain randomization will allow us to use less time on training by using simulation software while still producing robust controllers that adapt well into changing environments. With help from GPU-acceleration, we hope to speed up the process even further. Our findings suggest an improvement in transferability from models trained without domain randomization to models trained with domain randomization, as well as theoretical speed-ups from GPU-acceleration. At the batch sizes we ended up simulating the speedup was not enough to improve training amount substantially.**

***Index Terms*—Evolutionary Robotics, Domain Randomization, GPU-Acceleration, Sin-wave controller, PyGad, MuJoCo XLA.**

## I. Introduction

Simulation systems are popular in robotics as they allow rapid prototyping of physical systems, and enable data hungry machine learning approaches to become viable options for robotics control applications. Typically, this is done through various techniques such as reinforcement learning or evolutionary algorithms. These algorithms require extended periods of interaction between the algorithm and the environment for improvement. In many cases, carrying out the training in real life is too time and labour intensive to be feasible for many robotics learning tasks. Because of this, learning in simulation has become a central part of creating robotics controllers with deep learning.

The "reality gap" is a type of domain transferal problem which arises from the mismatch between training and application environment. Since the training is done in simulated environments, the learned behavior may be subtly (over)fitted to peculiarities of the simulator or factors which are different in real life. There have been many attempts at remedying this through various techniques, which we will mention briefly in the related work section.
Our methodology focuses on two techniques, namely domain randomization and GPU accelerated parallel simulation. The intent behind combining these techniques is to increase robustness through randomization, and make up for increased training requirements through parallelization of simulation.

## II. Related Work

### A. GPU - Acceleration

An excellent advancement in leveraging GPU acceleration for robotic physics simulations (e.g MuJoCo XLA [1,2]) is presented in the work by Liang et al.[3]. In comparison to conventional CPU-based simulations, the authors show considerable improvement in speed and scalability with their in-house physics simulator 'Nvidia Flex' coupled with single- and multi-GPU accelerated software. The authors show that complicated learning tasks, including teaching humanoid robots how to run, can be taught in the fraction of the time compared to previous works utilizing large clusters of CPUs. During their research they were able to teach a humanoid how to run in less than 20 minutes on a single GPU. This presents notable reduction in hardware needs for complex learning tasks.

### B. Domain Randomization

Domain randomization as a solution for the domain transfer problem is explored well in the paper by Tobin et. al[5]. In their paper, they use synthetic data generated in a digital 3D environment to teach a vision model how to locate objects on a table. This yields good results when transferring to a real environment.

Domain randomization is achieved by varying parameters of the training environment to achieve better generalization when

transferring domain. By introducing enough variability, the transfer may be seen by the model as ”just another variation”. In the paper they randomize things such as camera position, object texture and lighting.

### C. Using Obstacles to Promote Robust Locomotion

In the paper *Filling the Reality Gap - Using Obstacles to Promote Robust Locomotion for a Quadruped Robot* by A. Johnsen, he explores techniques for developing gaits that are more robust to reality transfer. The author claims that adding obstacles to the simulated environment is equivalent to adding noise, which should make the solutions found more robust. This is similar to the technique used in the aforementioned domain randomization paper, but on fewer parameters.
The physics simulator used in the paper is Nvidia PhysX. The obstacles are boxes of random dimensions placed around the testing environment.
The paper affirms that the technique yields positive results for transfer. This makes it an interesting question if a more comprehensive domain randomization would yield even better results. Randomizing parameters like gravity, friction and types of obstacle may allow the model to be even more robust towards reality transfer effects.

### D. Domain Randomization Techniques and Tanh-Wave Controller

Glette, Johnsen, and Samuelsen [6] propose a novel approach of mitigating the reality gap, this is achieved by incorporating obstacles within the simulated environment.

In the paper they performed simulations and experiments on a QuadraTot robot, which is a 4 legged ant model robot similar to the one we utilize in this paper. They used NVIDIA PhysX for simulation, where they used 5 different seeds that would change the position and size of boxes placed in front of the robot during simulation. For actuation control they employed a sinus-wave controller wrapped in a tanh function, the same we use for our experiments, the reason for the tanh wrapper is to ”flatten” the extremities of the wave-controller, which in turn may lead to more ground contact for the robot. Using this setup they performed experiments on the ant model.

Their findings suggest that evolving controllers through a randomized environment reduces the likelihood of producing controllers that are over-fit to smooth surfaces, which are often less transferable to real-life scenarios.

## III. Methodology

### A. Simulation Environment

We utilize MuJoCo[1] specifically MuJoCo XLA[2] to simulate and evaluate the performance of our controllers.
MuJoCo XLA enables GPU accelerated simulation through parallelization. GPU acceleration performs worse for single runs and certain scenes. GPU excels at running many instances of the simulation data in parallel using SIMD instructions. This is a natural fit for domain randomization as the variations can be spread across the instances. It is important to be aware of the trade-offs of GPU simulation, as it is not always a direct speed up. For complex scenes and single runs, CPU is substantially faster. Various simulation features are also unavailable in MJX.

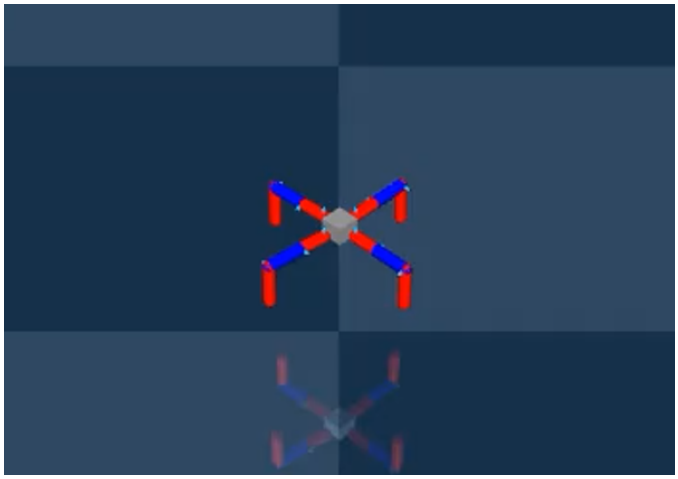


Fig. 1. The qutee robot model

*1) Regular MuJoCo:* MuJoCo allows loading models into the simulator through various means. We will use the XML ”qutee.xml” which contains the description of the makeup of the scene as well as the Qutee-robot. This file can also set various option parameters that affect the simulation like frictions, damping for joints, gravity and timestep size. After creating instantiating the model, data objects may be created from this model. The data objects contain simulation state and may be modified for each timestep of the simulation. We will be modifying the ctrl field of the data object according to values from the controller to move our robot. Other variables in data are qpos and xpos which allow us to see joint coordinate and cartesian coordinate locations of the various parts of the robot.
Passing the model and data object to mujoco.step() advances the simulation by one timestep. Advancing the simulation in a loop while updating the ctrl field between each step allows us to measure the performance of the robot by checking the distance from origin after a certain amount of time has elapsed.

*2) MuJoCo XLA:* To run simulations in parallel, one must create batches of the aforementioned data objects to be sent to the GPU. First, the objects are placed on the GPU. Using jax.vmap(), the mjx.step() function (same as mujoco.step() but for GPU usage) can be wrapped to accept batch arguments. This allows us to advance the simulation for all the data objects in parallel.

### B. Domain Randomization

For our testing we will randomize various simulator variables. In our case this is a slippage factor (impratio) for friction as well as gravity. This should make it so the most adept

controllers are less sensitive to variability of the aforementioned parameters. This is easily implemented thanks to native support in Mujoco XLA with JAX. This may affect speed however, as parameters mentioned reside within the model object created from the xml file. If the model for all the data objects is the same, it is possible to use a single one for simulation. Varying the parameters within the model requires creating batches of models in addition to batches of data. This may throttle the GPU by increasing the amount of data that resides on the GPU.

### C. Tanh-Wave Controller

The controllers will be represented by 4 floating point numbers that parameterize a sine wave through frequency, time shift, amplitude and offset. The output is sent through a tanh function [6] to prevent values outside the actuators range. These values will be evolved through the evolutionary algorithm. The controller has no feedback making it relatively ”dumb”, and we will account for this in our simulation setup as we can not expect it to adapt to complex situations.

$$X = A\tanh(\sin(2\pi tf + \phi) + d)$$

### D. Evolutionary Algorithm

We employ an Evolutionary Algorithm to optimize the parameters of the ant model controller. PyGAD[4] is a package that allow for efficient optimization of our parameters while also allowing for great customization of hyperparameters during selection, crossover and mutation techniques. During optimization we have set min- and max-values for each gene to correspond to the movement range of the Qutee-robot.

We take advantage of the ’on_start’ and ’on_generation’ methods to simulate the batch of robots in MJX, the results from the simulation is then passed on to the fitness function and used during selection.

*1) Population Initialization:* We initialize the population size to be as many robots we want to simulate concurrently, therefore this changes based on what experiments we would like to perform.

*2) Fitness Evaluation:* Fitness is evaluated based on the distance traveled by the model during simulation, which is passed into the EA and used for selection. So the further a solution has traveled in any direction, the higher fitness it gets.

For parent selection, mutation and crossover techniques we use built-in methods included in PyGAD[4]. Which is explained below.

*3) Selection and Reproduction:* For parent and survivor selection we make use of steady-state selection, which is also the default parent/survival selection technique in PyGAD[4]. However, this might change based on experiments we perform, others we might use are steady-state-selection or stochastic universal selection, we will specify if we use another selection than rank selection.

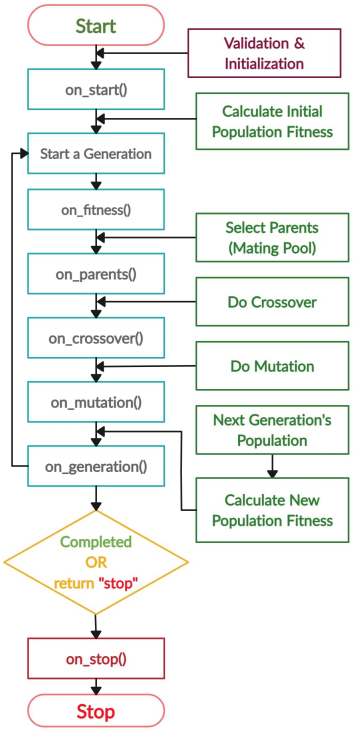



Fig. 2. Flowchart of PyGads functionality.

We use elitism to keep our most fit solutions in the population with K=5, meaning we keep the top 5 solutions, we choose to keep 5 solutions because we have such a large population.

*4) Mutation and Crossover:* During mutation we perform random mutation on the population. The mutation rate vary depending on what experiments we perform, we tend to use mutation rates between 0.1 to 0.25, which is the chance per gene to mutate, in which a relatively small random value gets added or subtracted from the previous gene.

We perform uniform crossover, which is performed by iterating through the 2 parents and selecting one gene to pass onto the children randomly chosen from the parents.

*5) Termination Criteria:* We perform simulation and evaluation of the robots for the number of generations we choose.

## EXPERIMENTS

### E. GPU vs CPU efficiency

We ran baseline tests to benchmark the performance of the CPU and GPU simulation. All GPU runs are done on an RTX3090, with the CPU runs being done on a Intel i7 8700K (6-core) processor. All CPU runs are single threaded, meaning additional CPU training time speedup could be achieved by running on multiple cores. We do not explore that in this paper, but it is easy to hypothesize gains based on multi core runs

since they should scale linearly.
We found CPU simulation to more than fast enough for real time simulation. Using a GPU for single runs was on the contrary very inefficient. To increase GPU efficiency, one must instantiate enough instances of data to maximize GPU utilization. Increasing instances once this threshold has been met will give diminishing (but still positive) returns on efficiency.

Using a batch size of around 4096 gives full utilization of the GPU. We used this size to allow faster testing while still benefiting from some speedup. More speedup is still possible by increasing the batch size beyond this number.

The baseline test is contrasted against a separate test where controller is trained with domain randomization while another is trained normally. These controllers are then tested in a separate environment where the friction of the ground and the gravity is modified to mimic an environmental transfer. The gravity is decreased from -9.81 to -8.81. The friction is changed from [1 0.005 0.0001] to [0.8 0.2 0.05]. The friction numbers affect various parts of the friction calculations which are further described on Mujoco's XML reference page.
As stated earlier, we perform batch simulation on the Qutee-model, which is a 4-legged ant model with ankle, knee, and hip joints.

### *F. Baseline Run*

In the baseline run we simulate a population of 4096 over 25 generations where we look at the best fitness over generations as well as comparing GPU to CPU runtime to see effects of bigger population sizes. Here we do not employ any kind of domain randomization, to further understand how well a genetic algorithm can optimize our controllers and set a standard for our next GA runs.

### *G. Baseline Run with Domain Randomization*

For our second run we will introduce some simple domain randomization techniques while still running a genetic algorithm with a population size of 4096 over 25 generations. Domain randomization techniques we are employing is randomizing gravity for each generation, as well as the friction coefficient. This is performed between each generation, so all necessary values for the domain randomization and the simulation is ready as we need them through the simulation process.

We hope that by adding randomization the controllers will be more robust and maybe lead to better performance.

## Results and Findings

### *H. GPU vs CPU Efficiency*

*1) Run times:* As seen when comparing figure 3 and 4, the time does not increase linearly when doubling the population size, which shows the advantage of using GPU compute for batch processing. It still runs slower than the same population size used for CPU processing. These CPU runs execute each

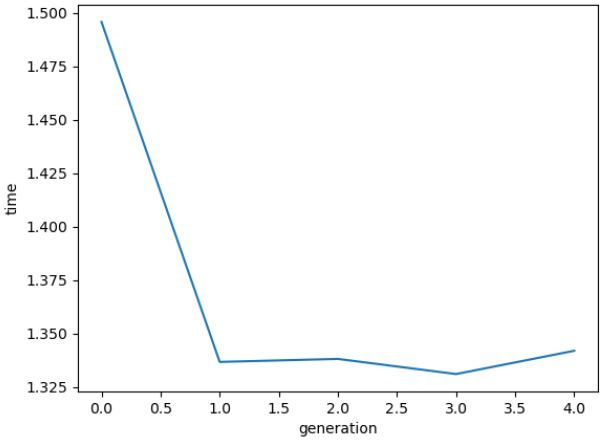



Fig. 3. 256 population size over 4 generations on GPU

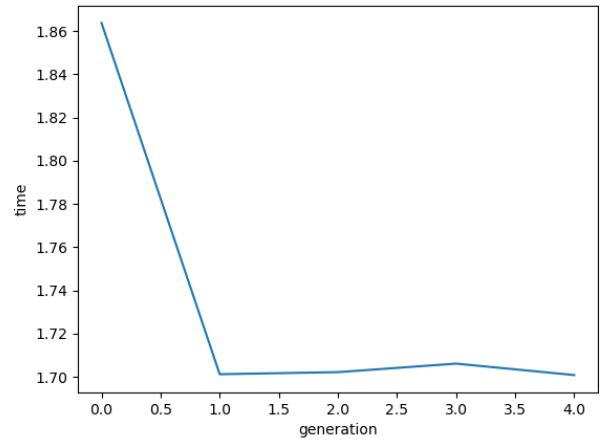



Fig. 4. 512 population size over 4 generations on GPU

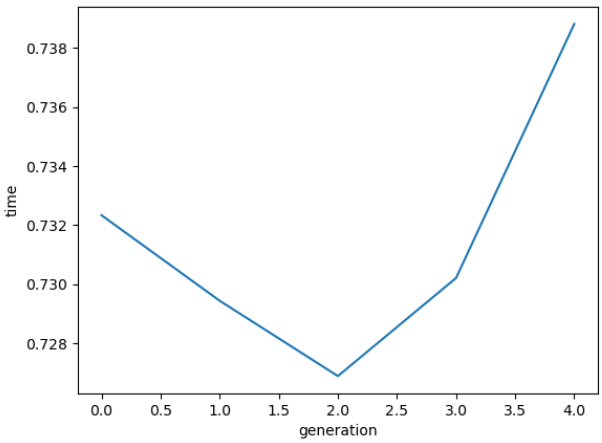



Fig. 5. 256 population size over 4 generations on CPU

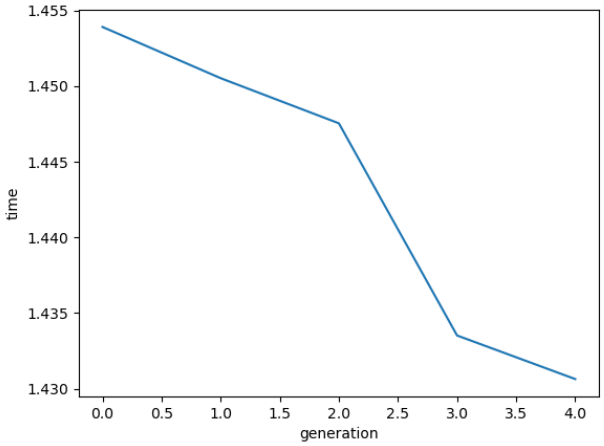



Fig. 6. 512 population size over 4 generations on CPU

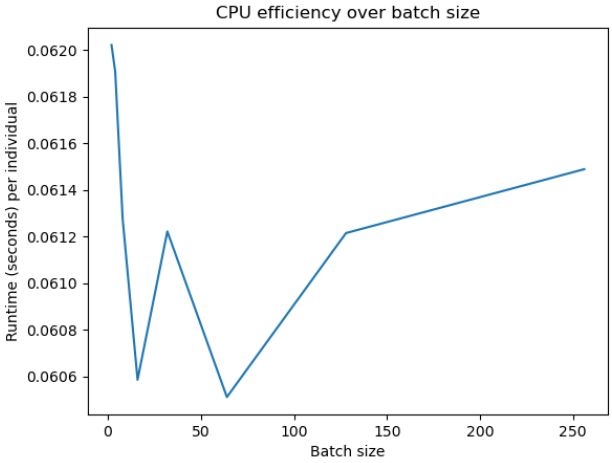



Fig. 8. Runtime per population with increasing population size on CPU

simulation sequentially. It is clear that CPU has the advantage at these population sizes. It is also shown that simulation time increases linearly with the population size for CPU simulation, as it doubles from 0.7 to 1.4 minutes. The final runtime per

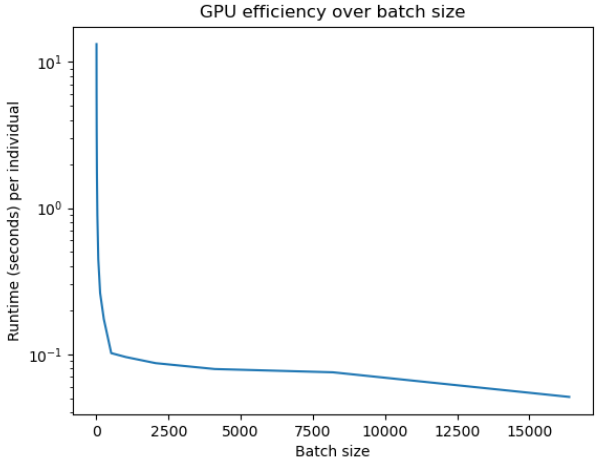



Fig. 7. Runtime per population with increasing population size on GPU

individual on a GPU batch size of 16384 is 0.05 seconds, which is similar to CPU run times which show a time of 0.061 seconds after running 256 simulations in sequence. Further profiling may be needed to understand what settings affect these times the most.

*2) Fitness gains:* Figures 9 and 10 show that both methods yield similar returns on fitness, meaning the simulator behaves similarly whether it is utilizes CPU or GPU.

### *I. Baseline Run*

Figure 9 shows the difference in runtime between the GPU and CPU simulations with a population size of 4096 over 25

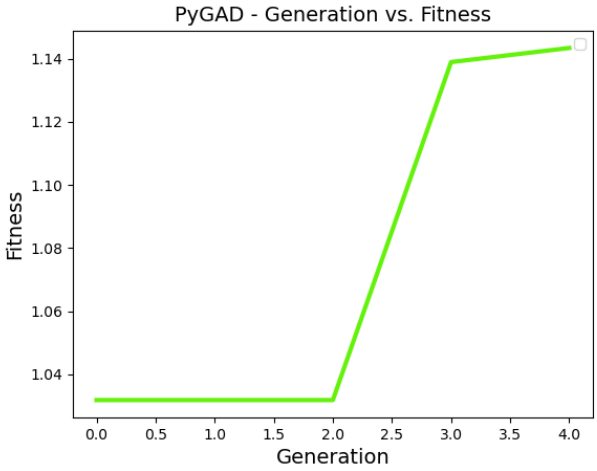



Fig. 9. 512 population size over 4 generations on GPU

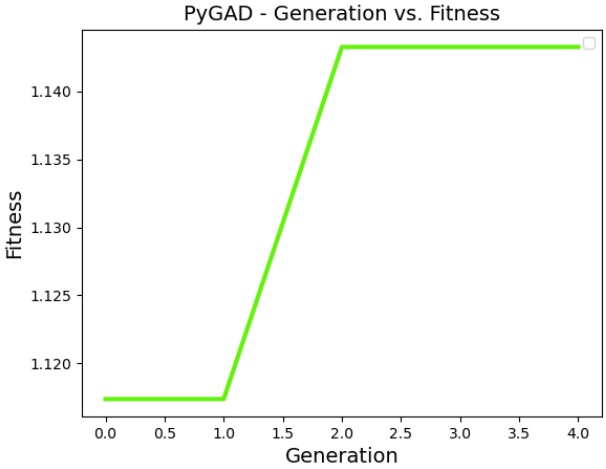
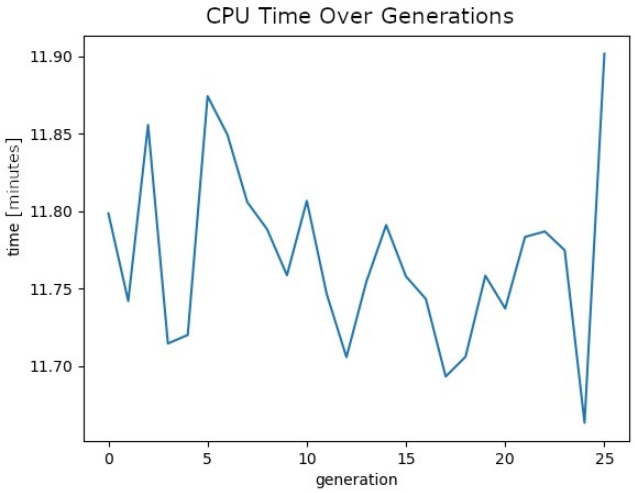



Fig. 10. 512 population size over 4 generations on CPU



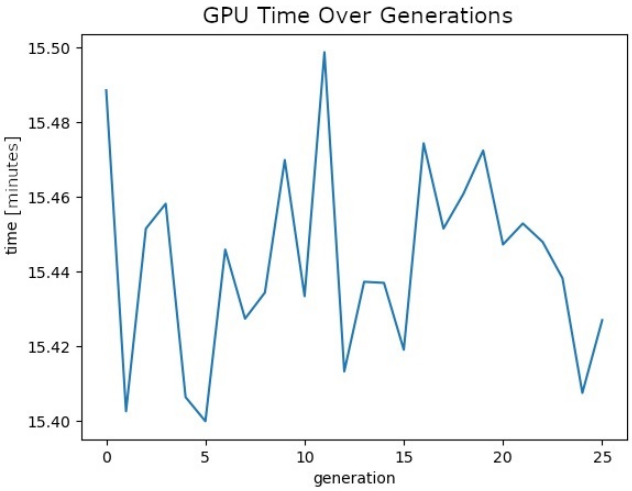




Fig. 11. Run times of GPU and CPU during baseline run

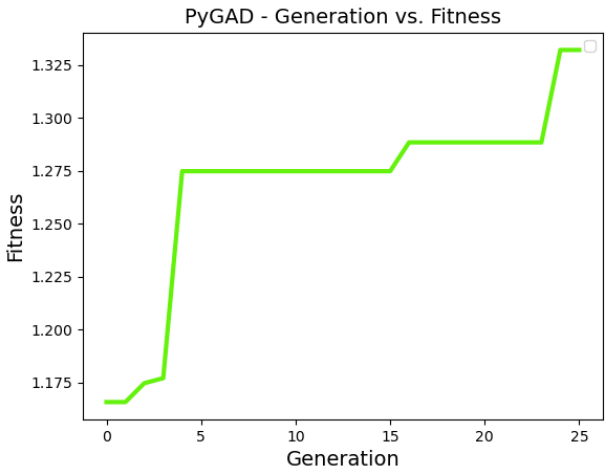



Fig. 12. Best fitness over generations, 4096 population over 25 generations.

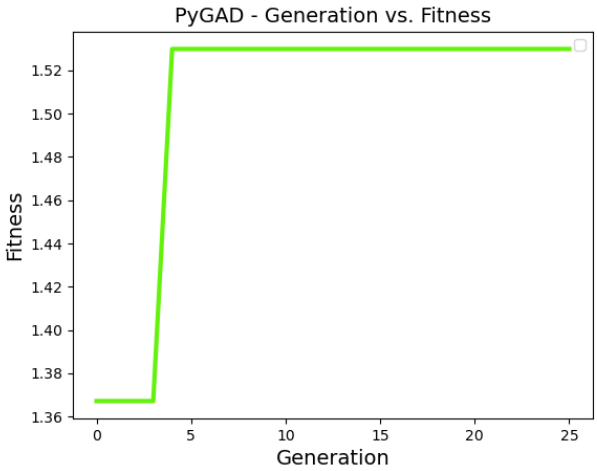



Fig. 13. Best fitness over generations, 4096 population over 25 generations with domain randomization.

generations.

We see a clear difference in run times when comparing GPU and CPU simulation times, the GPU using more time per generation, about 4 minutes difference. These results imply that the GPU does not perform to our expectations in this case. We have a few thoughts as to why, one of which being various simulator options not being optimized, the computation of too many contacts and too small batch size.

We do not get much insight through this fitness plot alone, we will look into the difference in fitness between the baseline run and a domain randomized one in the next subsection.

### *J. Baseline Run with Simple Domain Randomization*

Comparing figure 12 and figure 13 we see a better fitness from the run with domain randomization than the run without domain randomization. This is unexpected, as the non randomized controller should fit better to the environment it is evolved in. However, when we run the best controllers in the alternate environment, the randomized controller fares better. The non randomized controller achieves a fitness of 1.3 in the regular environment but only 0.2 in the altered environment. A visual inspection of the run shows the robot tips, possibly due to the gravity being lower and the controller not being accustomed to this. The randomized controller performs better in the alternate environment, improving the fitness to 1.7. This improvement may be luck, but the lack of decrease of performance informs us that the controller was more robust to the change.

## ETHICS STATEMENT

### *K. Ethics Introduction*

Our research is about reducing the reality gap of robots trained in simulation. The aim for this research is discovering new solutions, improve the training of robots in society and fellow researchers or students to explore a new method. However, this research also may be unethical due to requiring

a lot of environmental resources. We will therefore go into details about the ethical consideration and broader impacts of our research.

### L. Ethical Considerations

The energy consumption when training a robot is quite a lot. In our case especially the algorithm is intensive and requires a lot of computing power over a decent amount of time, it takes for example 5-20 hours of training for good results. We are aware that this could have a negative impact on the environment.

### M. Broader Impacts

Our research could help out societies by contributing to faster development of better autonomous robots that can take care of work for us, like for example repetitive work in the service industries. In addition, our research may contribute to developing robots that can do dangerous tasks, such as exploring environments with hazards, for us. How our research could contribute to these things is by publishing more knowledge about how to face the reality gap, thereby contributing to the training of robots around the world having fewer errors, having reduced costs, and taking less time. Our research may additionally contribute to new knowledge and insight in handling the reality gap when training robots, which might inspire researchers to explore the field further. This may also contribute to new methods or information being used in education, so that the next generation of students will have bigger head-start.

### N. Ethics Conclusion

Our research about reducing the reality gap when training robots could have a negative impact the environment due to the requirement of computational power. However, it may have many positive impacts on society. It may contribute to better robots being trained, which will help out making workplaces safer and more productive. On top of that, our research may lead to new findings on bridging the reality gap which might give researchers the push to delve more deeply into our proposed solution, as well as sharing more knowledge to the next generation of students on how to handle the reality gap when training robots.

## CONCLUSION

In this study we explored strategies to mitigate the ”reality gap” in robotics training by utilizing GPU acceleration and domain randomization techniques. Our results show the significant trade-offs between GPU and CPU-based simulation, demonstrating that while GPU parallelization offers great scalability, the efficiency of GPU-based simulation depends on optimizing batch sizes and data handling as we do not meet our expectations because of this. Furthermore, our findings suggest that domain randomization generated more ”realistic” and robust controllers, which enabled better performance in altered environments.

Utilizing these methods, we manage to combine evolutionary robotics with domain randomization and GPU acceleration to improve simulation-to-reality transferability, and has laid ground for further work with these methods. Despite the computational demands of our approach, the future of this research has potential of developing safer and more versatile robots. We hope this work inspires further innovations in bridging the reality gap, contributing to the development of robust robots able to adapt to changing environments.

## ACKNOWLEDGMENTS

We would like to extend our gratitude to our supervisors, Kai Ellefsen and Kyrre Glette, as their guidance, support and feedback made our research possible. We also extend our thanks to Lazo Omar and Ahmed Murad for their close cooperation throughout our endeavour. Working together, with regards to the similarity of our research, was of tremendous help.

## REFERENCES

[1] Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, 5026–5033., 10.1109/IROS.2012.6386109
[2] MuJoCo XLA
[3] Jacky Liang, Viktor Makoviychuk, Ankur Handa, Nuttapong Chentanez, Miles Macklin, Dieter Fox, GPU-Accelerated Robotic Simulation for Distributed Reinforcement Learning, arXiv:1810.05762 [cs.RO], 2018
[4] Gad, Ahmed Fawzy, Pygad: An intuitive genetic algorithm python library, 2023, Multimedia Tools and Applications, 1-14., Springer, GitHub
[5] Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017, March 20). Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World.
[6] K. Glette, A. L. Johnsen and E. Samuelsen, ”Filling the reality gap: Using obstacles to promote robust gaits in evolutionary robotics,” 2014 IEEE International Conference on Evolvable Systems, Orlando, FL, USA, 2014, pp. 181-186, doi: 10.1109/ICES.2014.7008738. keywords: Joints;Legged locomotion;Robustness;Noise;Knee;Servomotors,