IOWA STATE UNIVERSITY

Plant Manipulation Work

Professor Soumik Sarkar, Self-Aware Complex Systems Laboratory

Felipe Leguizamo

Fall 2022

Table of Contents

Abstract

The purpose of this document is to establish a full record of what has been done on the plant manipulation work in Fall 2022. The goal of this project is to train a deep reinforcement learning (DRL) agent to manipulate a deformable plant. The agent shall seek to clear an occlusion of a fruit or vegetable by the plant, while avoiding damage to the plant. The agent can interface with the environment by commanding a robotic manipulator. Several simulated environments were created to develop an DRL agent capable of completing this task.

Low Fidelity Simulation

The first step in this project was to develop low-fidelity simulations that could be used to rapidly try out state spaces, action spaces, and reward functions. The results of these simulations are summarized below and were presented at MLCAS 2022 (1).

Link Model

The first set of simulations was performed using a kinematic model of a plant. The stem was modeled as a series of links and joints, which were represented with Denavit-Hartenberg parameters. The DRL agent was then able to manipulate each joint by rotating it a set angle in the positive or negative direction.

Table 1: DRL formulation for link model trainings

State space	$s = [\theta_0, x_0, y_0, \theta_1, x_1, y_1, ...\theta_n, x_n, y_n]$ n is the number of joints, θ, x, y represent the angle, x position, and y position of a joint.
Action space	$a_t = [-n, -n+1, -n+2, ...n-2, n-1, n]$ There are 2n discrete actions, with the sign representing the direction a manipulator will rotate a joint.
Reward function	$r_t = \alpha + \gamma$ $\alpha = -10$ if greater than 3 manipulations are performed on a joint; $\gamma = 300$ if the occlusion is cleared

Simulations conducted using the plant model indicated that it was relatively easy to converge onto a satisfactory policy:

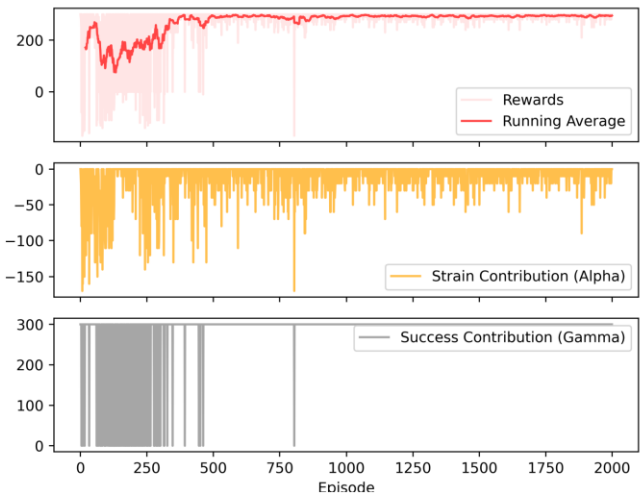

Figure 1: Results of training on link model

Sample code using DQN and PFRL is in the **plant_rl_demo.py** script in the repo. Note that in this sample script, an additional term β was added to the reward function to penalize total manipulations.

The purpose of this simplified simulation was to rapidly experiment with different observation spaces, action spaces, and reward functions. It can be leveraged for this purpose again in the future, but there are higher fidelity simulation options that should be pursued instead.

Cantilever Beam Model

The next set of simulations was performed using a cantilever beam model of a plant. From beam deflection equations, the following model can be utilized to simulate a plant with a given modulus of elasticity (1):

$$y = -\frac{Fx^2}{6EI}(3a - x) \qquad 0 < x < a$$

$$y = -\frac{Fa^2}{6EI}(3x - a) \qquad a < x < l$$

Where x and y are the coordinates of a continuous plant stalk, and a is the location of the force application measured from the base:

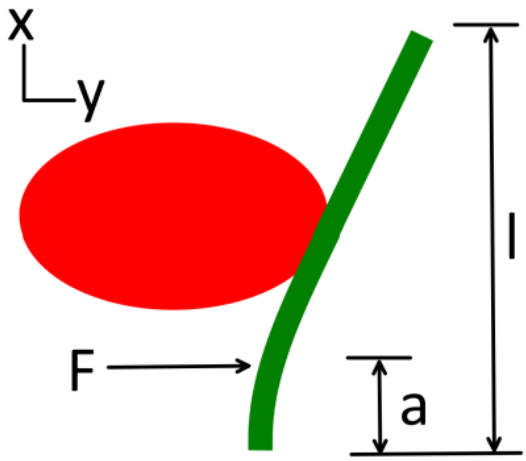

Figure 2: Cantilever beam model of a plant, with associated reference frame

In the code, the plant stalk is discretized, and its position is represented with an array. The DRL agent is then allowed to interface with the environment:

Table 2: DRL formulation for cantilever beam model trainings

State space	$s_t = [x, y, \nu, x_f, y_f, f_r, F]$ x and y are vectors representing the current position of points on the cantilever beam, ν is the occlusion of the fruit, x_f and y_f are the x and y position of the fruit center, f_r is the radius of the fruit, and F is the force applied to the plant.
Action space	$a_t = [\Delta F, \Delta a]$ ΔF is a change in force applied to the plant, and Δa is a change in force location along the plant.
Reward function	$r_t = \alpha + \beta + \gamma$ $\alpha = -750\sigma_{max}^2$ with σ_{max} equaling the maximum Von Mises stress in the beam, $\beta = -100\nu$, $\gamma = \lvert r_t \rvert + 500$ if the occlusion is cleared; $\gamma = -2000$ if σ_{max} exceeds 75 MPa. Assume 90 MPa yield stress for the stem.

This formulation allows for a DRL policy to be trained:

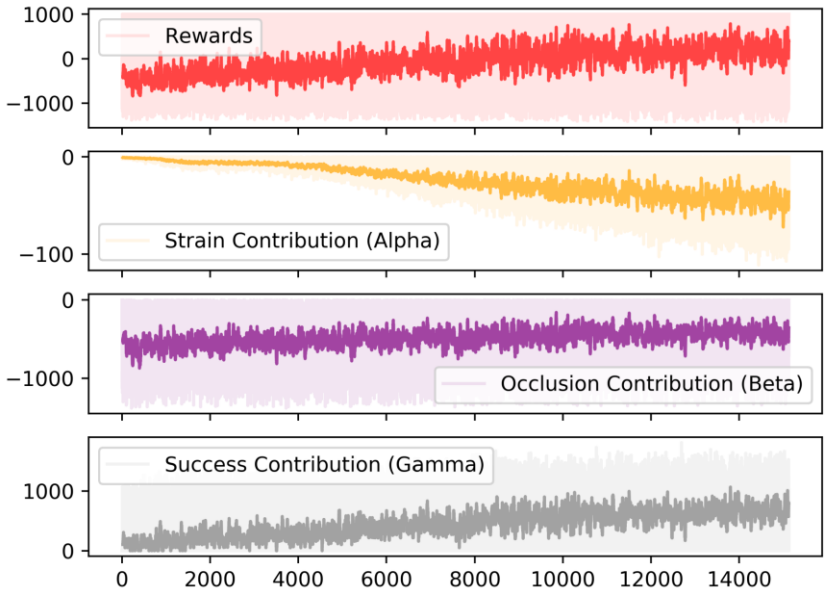

Figure 3: Results of training on the cantilever beam model

Notably, the success contribution (γ) did not reach 1000 every single time. This is actually a favorable outcome: the fruit position and size was randomized for each episode. This led to some possibilities where the fruit was close to the stem, making it impossible to clear the occlusion without breaking the plant:

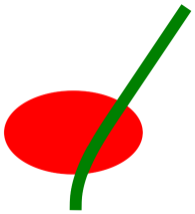

Figure 4: Unsolvable case. Clearing the occlusion would require too much stress at the base

This leads to a necessary requirement for any DRL policy that manipulates a plant: the agent must be able to identify cases where it is not possible to clear the occlusion. In these cases, it should take no action.

In the repo, an example with PFRL and DQN is available (**plant_beam_rl_demo.py**), as well as a Stable Baselines implementation with PPO and SAC (**plant_beam_rl_ppo_demo.py** and **plant_beam_rl_sac_demo.py**).

In my opinion, the cantilever beam model is more useful for prototyping reward functions than the link model. I think it provides a nice alternative to what the Carnegie Mellon team has done (which is to model a plant as a series of links with spring/damping coefficients). Potential future work could include finding a way to model the cantilever beam in 3D. A key advantage of this is that it would likely be more computationally efficient than a finite element model, while still maintaining a reasonable level of accuracy.

High Fidelity Simulation: Isaac Gym

Isaac Gym Introduction

The motivation for using Isaac Gym came from a discussion with the Carnegie Mellon team. They seem to have some success modeling plants as a series of links with spring/damping coefficients. This method is backed by experiments that show that the mechanics of a real-world plant resemble the dynamics of their spring/dashpot model enough to train a DRL policy on it.

Isaac Gym comes with additional benefits:

1. It has the capability to simulate parallel environments.
2. It comes with two physics engines (PhysX and Flex).
3. The PhysX engine can be run on a GPU, meaning that both the physics model and the ML pipeline can be optimized for performance.
4. The Flex engine supports deformable body simulations using an FE solver.

Isaac Gym Setup and Support

Instructions for installing Isaac Gym are here. It is necessary to sign up with a free NVIDIA developer account. For user support, I have found that the forums are not very responsive – I suspect that the team at NVIDIA working on Isaac Gym is small and can't keep up with the huge number of researchers that have shown interest in this tool. However, there are usually just enough answers that the search feature can be useful for common questions. Finally, I have found that it is sometimes useful to search GitHub for specific Isaac Gym functions to see how they are used in an example. The standard documentation that comes with the software often leaves out critical information about arguments to functions, what gets returned, how to handle tensors, etc.

Experiments

I used Isaac Gym to simulate a deformable body that is placed in front of a robot (Franka Panda). This object blocks a fruit, and the task for the robot arm is to clear the occlusion of the fruit.

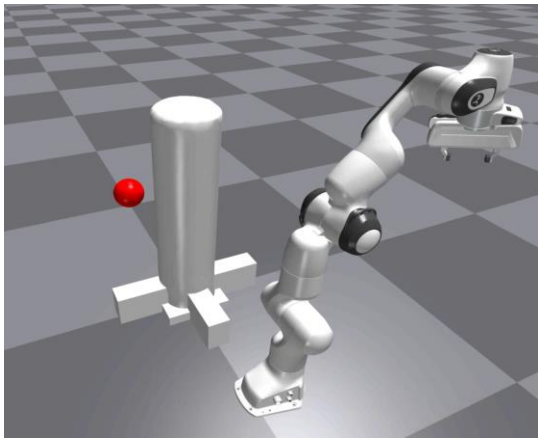

Figure 5: Deformable cylindrical object blocking the way between a robot and a fruit. Note the prismatic supports at the base to fix the deformable body in place. These were added after the deformable base began to permanently deform after interaction with the manipulator

An OpenAI Gym environment is defined to allow the agent to manipulate the environment via the Franka arm:

Table 3: DRL formulation for Isaac Gym Trainings

State space	$s_t = [g_1, g_2 \dots g_{1600}]$ where g_n is a grayscale pixel value in a 40x40 image
Action space	$a_t = [-n, -n+1, -n+2 \dots n-2, n-1, n]$ where n is the number of joints. The positive or negative sign determines whether the joint is moved a positive or negative amount $d\theta$
Extrinsic reward function	$r_t = -5$ if any joint is within $3d\theta$ of hardware limits (defined in URDF) $r_t = -5$ if joint 0 is at $\theta < 0.1$ $r_t = 10$ if occlusion of fruit has been cleared

Note that the camera is placed above the arm, facing towards the location of the deformable body.

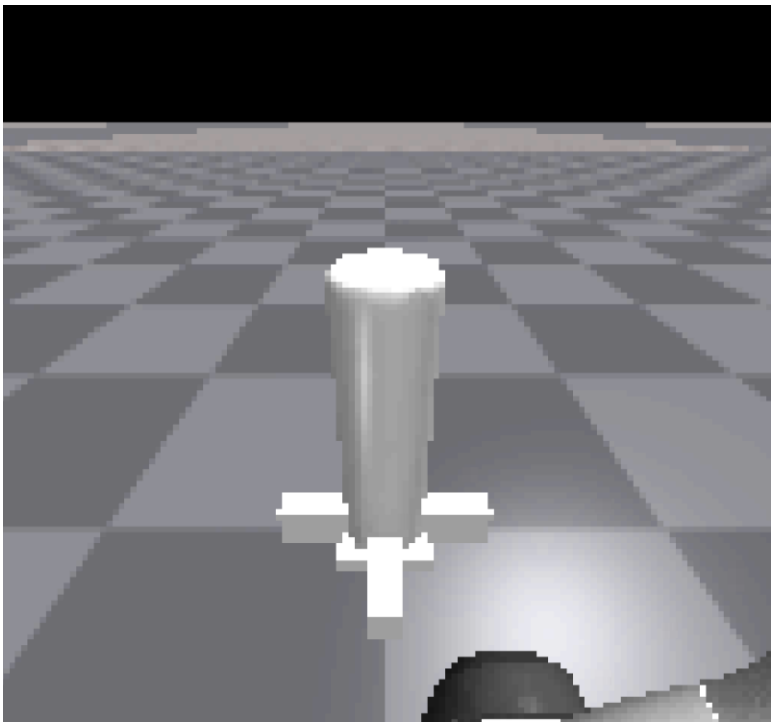

Figure 6: Sample camera output. This is downsampled to a 40x40 image and converted to grayscale before being provided to the agent as the state space.

Note that the extrinsic reward function is crafted for semi-sparse rewards. There is a negative reward assigned to the case where a joint is within $3d\theta$ of a hardware limit, which is the maximum angle a joint can reach. There is also a negative reward assigned when the base (joint 0) is in a certain range (less than 0.1 radians). This is to encourage the agent to explore the state space directly in front of it.

Finally, an intrinsic curiosity model (ICM) is implemented as originally described in the paper by Pathak et al (2). I found a really nice YouTube tutorial by Phil Tabor that helped me with the bulk of the PyTorch setup (3). I highly recommend it to anyone who has read the paper and is interested in implementing it. I also highly recommend reading over Thomas Simonini's blog post on the subject: it is a wonderful gentle introduction to curiosity (5).

Phil's video uses A3C, which I think is perfectly suited for the Isaac Gym environment. A3C utilizes multiple environments and multiple workers to update a local network (6). These local networks are then used to update a global network, which is the result after the training is complete. In Isaac Gym, there is the capability to simulate many environments in a grid, as shown in the following figure:

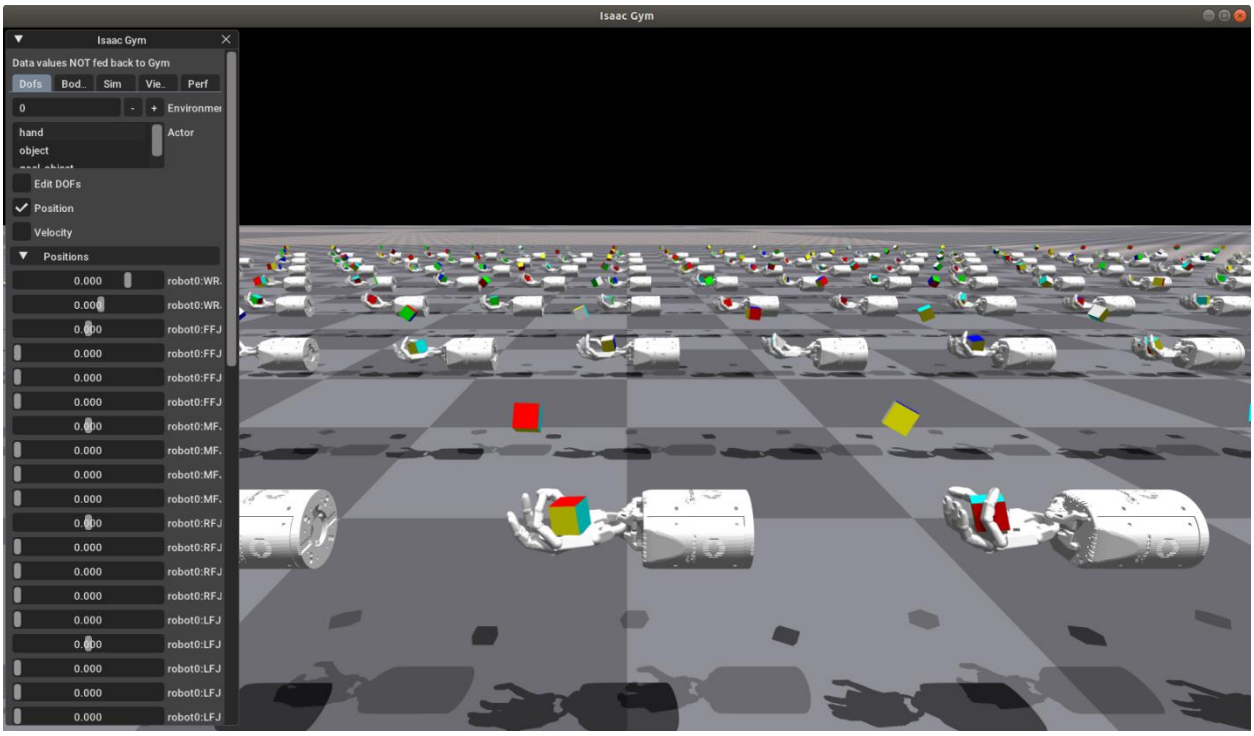

Figure 7: A grid of robotic hands learning to manipulate a cube in Isaac Gym

Because of this, the structure of A3C makes perfect sense for an environment like the one in this work. However, I ran into some issues when trying to implement this:

1. My desktop PC was not powerful enough to handle more than ~7-10 environments. I ended up with a CUDA error indicating I ran out of memory.
2. I kept running into PyTorch errors when creating a list of networks for each environment. See **parallel_env.py**:

```
envs = []
env_local_agents = []
env_memories = []
env_ep_done = []
env_hx = []
scores = []*n_envs
ep_counters = []
env_obs = [None]*n_envs
local_icms = []

for i in range(0, n_envs):
    envs.append(IsaacGymPlantEnv(simulation, i, observation_mode='Grayscale Image',
action_mode='All Joints'))
    env_local_agents.append(ActorCritic(input_shape, n_actions))

    if icm:
          local_icms.append(ICM(input_shape[0], n_actions, intrinsic_gain=1/4000))
```

Here, I am attempting to create a list of networks (the ActorCritic and ICM objects) for all environments in my Isaac Gym simulation. If you run this with more than one environment, you will run into an error. Therefore, I think it is worthwhile for future work to focus on improving the parallelization of environments.

Despite these challenges, the simulation works perfectly well with one environment. The following parameters were used in the ICM setup:

Table 4: ICM parameters, including optimizer

Parameter	**Value**
η (Intrinsic Gain)	0.00625
T_{max} (how many episodes in between ICM updates)	20
Learning rate (Adam Optimizer)	1e-4

Results

Right before break, I ran trainings in Isaac Gym to try and evaluate the effect of the ICM. This is a simple true/false flag in the highest-level script, so it is easy to clone the repo on two separate machines and attempt the training.

Before finals week, I demonstrated that using the ICM provided a performance improvement when training in Isaac Gym using the formulation in Table 3. The agent was able to manipulate the plant to clear the occlusion and reach higher rewards quicker. However, I noticed that the base of the deformable object was not fixed well, leading to some permanent deformation that I felt biased the results.

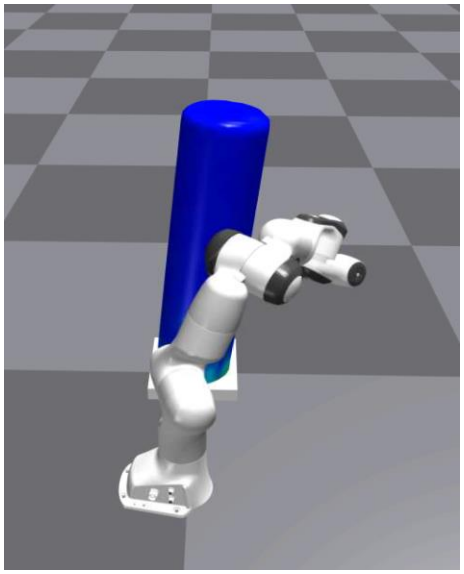

Figure 8: Deformation caused by manipulation. Note that the deformable object is permanently bent to the right even though it is not directly being manipulated.

This problem is illustrated in Figure 8, and it became problematic if the deformation led to the fruit being visible at all times, leading to a successful occlusion being recorded without any actual robot interaction. I added reinforcements to the base of the object, as shown in Figure 5, and this seemed to address the issue.

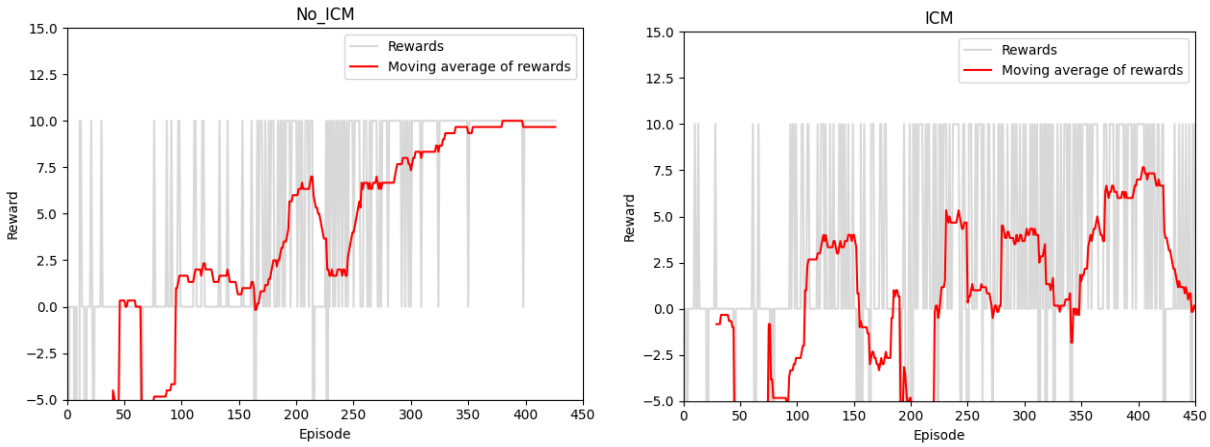

Figure 9: Training in Isaac Gym without curiosity (left) and with curiosity (right)

The results in Figure 9 show the trainings after the base reinforcements were added. It is somewhat disheartening to see that the intrinsic reward did not accelerate the time it took to reach consistent rewards. I suspect that the agent might explore past the point when it should stop. I also believe that this task was not too complicated. Without curiosity, and with nothing but the camera image and control of the joints, the agent was able to consistently clear the occlusion (reward of 10) with no issue in about 400 episodes. Curiosity exists to seek out extremely sparse rewards, so that makes me wonder if the benefits of the ICM are better seen with a more complex task. I will discuss this more in the Future Works section.

Still, I am intrigued by my initial results where the ICM-guided trainings led to earlier successes. Even if this was not as robust an environment as what is shown in Figure 9 (after a while, the robot would deform the object so much that the fruit was always visible), it is still the same environment for both the ICM and non-ICM trainings. The fact that curiosity was able to accelerate the agent's learning in this scenario is encouraging.

Codebase

The code is kept in a GitHub repo tied to my personal account (7). Here is an overview of what it contains:

| **.gitignore** -> File with list of filetypes/directories not to be pushed to git
| **franka_joint_eval.py** -> Test script that plots transient joint motions to a target. Use to debug control issues in Isaac Gym.
| **isaacgym_rl_curiosity_demo_a3c.py** -> The main script used for trainings in the Results section above. Recommend to start here, and then look into **parallel_env.py** below for the bulk of the code.
| **isaac_gym_demo.py** -> No RL in here, but a good intro to how Isaac Gym startup works. I intentionally wrote the Simulation class to try and make this easier for future development.
| **parallel_env.py** -> The bulk of the code for ICM training. This will be where most future development will focus. It is intended and written to be parallelized, but I have gotten Pytorch errors when trying do so. Therefore, I only ran with one environment in Isaac Gym.
| **plant_beam_rl_demo.py** -> Demo of RL training using the cantilever beam model and DQN.
| **plant_beam_rl_ppo_demo.py** -> Demo of RL training using the cantilever beam model and PPO.

| **plant_beam_rl_sac_demo.py** -> Demo of RL training using the cantilever beam model and SAC.
| **plant_rl.yml** -> List of conda and pip dependencies. I suggest creating a conda environment from this file and running Isaac Gym from the activated environment.
| **plant_rl_demo.py** -> RL code for link model.
| **plant_rl_variable_init.py** -> RL code for link model, with the position of the links randomized every episode.
| **plot.py** -> Plotting functions. They take as input the CSV logs generated by the Isaac Gym trainings.
+---actor_critic
| **__init__.py** -> Actor critic implemented in Pytorch. See Phil Tabor's videos for more details (3).
|
+---archive -> A bunch of deprecated demonstrations. You can skip this directory.
| | **isaacgym_rl_curiosity_demo_PPO.py**
| | **model_demo.py**
| | **plant_beam_rl_icm_demo.py**
| | **test_model.py**
| | **worker.py**
| |
| **\---ICM_Demo**
| | **icm.py**
| | **icm_init.py**
| |
| **\---ICM_Plant**
| **icm.py**
| **icm_init.py**
| **__init__.py**
|
+---basic_model -> Source code for simplified link and cantilever beam models, as well as the associated Gym environments.
| **kinematics.py**
| **plant_beam_model.py**
| **plant_beam_model_continuous_env.py**
| **plant_beam_model_env.py**
| **plant_model.py**
| **plant_model_env.py**
| **point.py**
| **__init__.py**
|
+---ICM -> Source code for the ICM. See Phil Tabor's videos for more information (3).
| **memory.py**
| **__init__.py**
|
+---isaacgym_sim -> Code for Isaac Gym simulation and Gym wrapper.
| **isaacgym_env.py** -> Isaac Gym environment. Highly suggest going through this to understand what happens at each step in the training.
| **__init__.py** -> Simulation class I wrote. Simplifies the user's interface with Isaac Gym into easy commands for tasks like moving joints, stepping physics, working with the FE solver, etc.
|

+---**optimizer**
| **__init__.py** -> Adam optimizer. See Phil Tabor's videos for more information (3).
|
\---**out** -> Output directory for logs. Has an empty file so that the empty directory can be pushed to git.
gitplacehold

Future Work

I'd like to preface this section by saying that Isaac Gym is a very powerful and exciting tool. I see it as something that could benefit a lot of robotics projects soon. The capabilities are being expanded slowly but surely, and the CMU team has had some successes with it. There are many directions that I would take this work in the future.

First, I would try to get worker parallelization working with Pytorch. This will be critical going forward. As I mentioned, the current tasks are somewhat simple, but as they become more complex and difficult to train, a good parallelization scheme should help accelerate the process. I think the CMU team demonstrated this, so they could be a good resource to ask.

Second, I would try to see whether a more complex task brings out the benefits of using the ICM. In Figure 9, the task might have been simple enough to accomplish without curiosity anyways. However, what about a very difficult task with even sparser rewards? What if the robot had to manipulate the plant, then uncover a vegetable that is underground (like a potato or a carrot)? If the scenario is difficult enough, I could see a possibility where almost no amount of training (without curiosity) would result in success. Perhaps here, the benefits of curiosity could come to light.

Third, I have a few ideas for utilizing the FE solver provided to us by Baskar's group. We know that the FE engine in Isaac Gym struggles to realistically model thin rods (such as corn stalks), so it could be worthwhile to return to this tool in the future. I remember a concern being that the code is in C and porting over the data to Python could be a challenge, but I have some ideas here as well. Python supports using C data types and passing them to DLLs, and this is something I have done before. Therefore, I imagine that it wouldn't be too difficult to set up an FE engine using the C code, and then bringing it into the loop of a Gym environment. If there is a point in the future where the lab could benefit from this kind of simulation environment, please let me know and we can talk more about paths forward.

Suggested Best Practices

Running the code is simple:

1. Install Isaac Gym (8)
2. Clone the repo (7)
3. Set up the conda environment using the included .yml file in the repo
4. Run the desired script

Both my machine and the machine in the robotics lab have this setup ready to go. I highly suggest continuing to use git for good version control, especially if more than one person is working on this project. Even just working on this myself, careful use of git saved a lot of headaches when the code started getting more complicated. Conda environments are also a great idea, since I've noticed that specific versions of Pytorch, Numpy, and Pandas can cause dependency issues.

Finally, please don't hesitate to contact me if there are questions that come up. I can almost guarantee that there will be since Isaac Gym comes with a bit of a learning curve and NVIDIA's documentation and forums are not great. I'd much rather devote some time even after I am graduated than have the effort I put into this project be for naught. My contact info is below, and I am always happy to set up a Zoom call or do a walkthrough of the code I wrote:

leguizamofelipe@outlook.com

515-318-2692

References

1. Leguizamo, D. F., Chawla, Y., Saurabh, K. et. al. Robotic Manipulation for Plant Interaction Using Reinforcement Learning. Poster presented at MLCAS 2022.
2. Al-Zube, L., Sun, W., Robertson, D. et al. The elastic modulus for maize stems. Plant Methods 14, 11 (2018). https://doi.org/10.1186/s13007-018-0279-6
3. Deepak Pathak, Pulkit Agrawal, Alexei A. Efros and Trevor Darrell. Curiosity-driven Exploration by Self-supervised Prediction. In ICML 2017.
4. https://www.youtube.com/watch?v=O5Z-3q-J18I
5. https://towardsdatascience.com/curiosity-driven-learning-made-easy-part-i-d3e5a2263359
6. https://medium.com/@dmonn/how-does-an-a3c-work-4e02266d1a96
7. https://github.com/leguizamofelipe/plant_rl
8. https://developer.nvidia.com/isaac-gym