GPS Documentation

Release 0.1

Chelsea Finn, Marvin Zhang, Justin Fu, Zoe McCarthy, Xin Yu Tan

February 23, 2016

Contents

CHAPTER 1

Guided Policy Search

This code is a reimplementation of the guided policy search algorithm and LQG-based trajectory optimization, meant to help others understand, reuse, and build upon existing work. It includes a complete robot controller and sensor interface for the PR2 robot via ROS, and an interface for simulated agents in Box2D and Mujoco. Source code is available on GitHub.

While the core functionality is fully implemented and tested, the codebase is **a work in progress**. See the FAQ for information on planned future additions to the code.

1.1 Relevant work

Relevant papers which have used guided policy search include:

- Sergey Levine*, Chelsea Finn*, Trevor Darrell, Pieter Abbeel. *End-to-End Training of Deep Visuomotor Policies*. 2015. arxiv 1504.00702. [pdf]
- Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, Pieter Abbeel. *Learning Deep Neural Network Policies with Continuous Memory States*. ICRA 2016. [pdf]
- Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, Pieter Abbeel. *Deep Spatial Autoencoders for Visuomotor Learning*. ICRA 2016. [pdf]
- Sergey Levine, Nolan Wagener, Pieter Abbeel. *Learning Contact-Rich Manipulation Skills with Guided Policy Search*. ICRA 2015. [pdf]
- Sergey Levine, Pieter Abbeel. *Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics*. NIPS 2014. [pdf]

If the codebase is helpful for your research, please cite any relevant paper(s) above and the following:

- Chelsea Finn, Marvin Zhang, Justin Fu, Xin Yu Tan, Zoe McCarthy, Emily Scharff, Sergey Levine. Guided Policy Search Code Implementation. 2016. Software available from rll.berkeley.edu/gps.

For bibtex, see this page.

1.2 Installation

1.2.1 Dependencies

The following are required

- python 2.7, numpy (v1.7.0+), matplotlib (v1.5.0+), scipy (v0.11.0+)
- boost, including boost-python
- protobuf (apt-get packages libprotobuf-dev and protobuf-compiler)

One or more of the following agent interfaces is required. Set up instructions for each are below.

- Box2D
- ROS
- Mujoco

One of the following neural network libraries is required for the full guided policy search algorithm

- Caffe (master branch as of 11/2015, with pycaffe compiled, python layer enabled, PYTHONPATH configured)
- TensorFlow (coming soon)

1.2.2 Setup

Follow the following steps to get set up:

1. Install necessary dependencies above.
2. Clone the repo:

```
git clone https://github.com/cbfinn/gps.git
```

3. Compile protobuffer:

```
cd gps
./compile_proto.sh
```

4. Set up one or more agents below.

Box2D Setup (optional)

Here are the instructions for setting up Pybox2D.

1. Install Swig and Pygame:

```
sudo apt-get install build-essential python-dev swig python-pygame subversion
```

2. Check out the Pybox2d code via SVN

```
svn checkout http://pybox2d.googlecode.com/svn/trunk/ pybox2d
```

3. Build and install the library:

```
python setup.py build
sudo python setup.py install
```

Mujoco Setup (optional)

In addition to the dependencies listed above, OpenSceneGraph(v3.0.1+) is also needed.

1. Install Mujoco (v1.22+) and place the downloaded `mjpro` directory into `gps/src/3rdparty`. Mujoco is a high-quality physics engine and requires requires a license. Obtain a key, which should be named `mjkey.txt`, and place the key into the `mjpro` directory.
2. Build `gps/src/3rdparty` by running:

```
cd gps/build
cmake ../src/3rdparty
make -j
```

3. Set up paths by adding the following to your `~/.bashrc` file:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:gps/build/lib
export PYTHONPATH=$PYTHONPATH:gps/build/lib
```

   Don't forget to run `source ~/.bashrc` afterward.

ROS Setup (optional)

1. Install ROS, including the standard PR2 packages
2. Set up paths by adding the following to your `~/.bashrc` file:

```
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:/path/to/gps:/path/to/gps/src/gps_agent_pkg
```

   Don't forget to run `source ~/.bashrc` afterward.

3. Compilation:

```
cd src/gps_agent_pkg/
cmake .
make -j
```

WARNING - Our controller directly commands torques on the PR2 robot with no safety measures. Closely supervise the robot at all times, especially when running code for the first time.

ROS Setup with Caffe (optional)

This is required if you intend to run neural network policies with the ROS agent.

1. Run step 1 and 2 of the above section.
2. Checkout and build caffe, including running `make -j && make distribute` within caffe.
3. Compilation:

```
cd src/gps_agent_pkg/
cmake . -DUSE_CAFFE=1 -DCAFFE_INCLUDE_PATH=/path/to/caffe/distribute/include -DCAFFE_LIBRARY_PAT
make -j
```

   To compile with GPU, also include the option `-DUSE_CAFFE_GPU=1`.

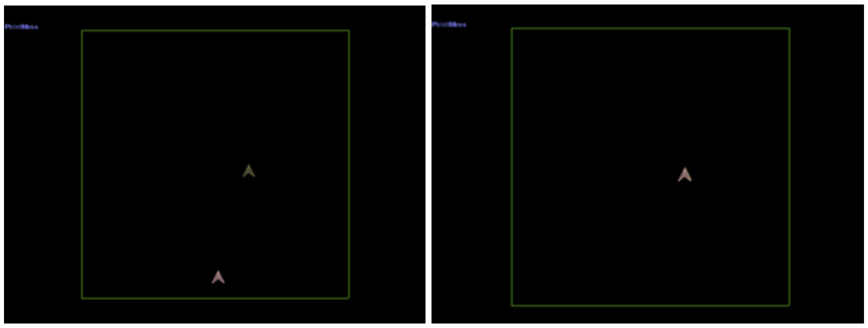
1.3 Examples

1.3.1 Box2D example

There are two examples of running trajectory optimizaiton using a simple 2D agent in Box2D. Before proceeding, be sure to *set up Box2D*.

Each example starts from a random controller and learns through experience to minimize cost.

The first is a point mass learning to move to goal position.

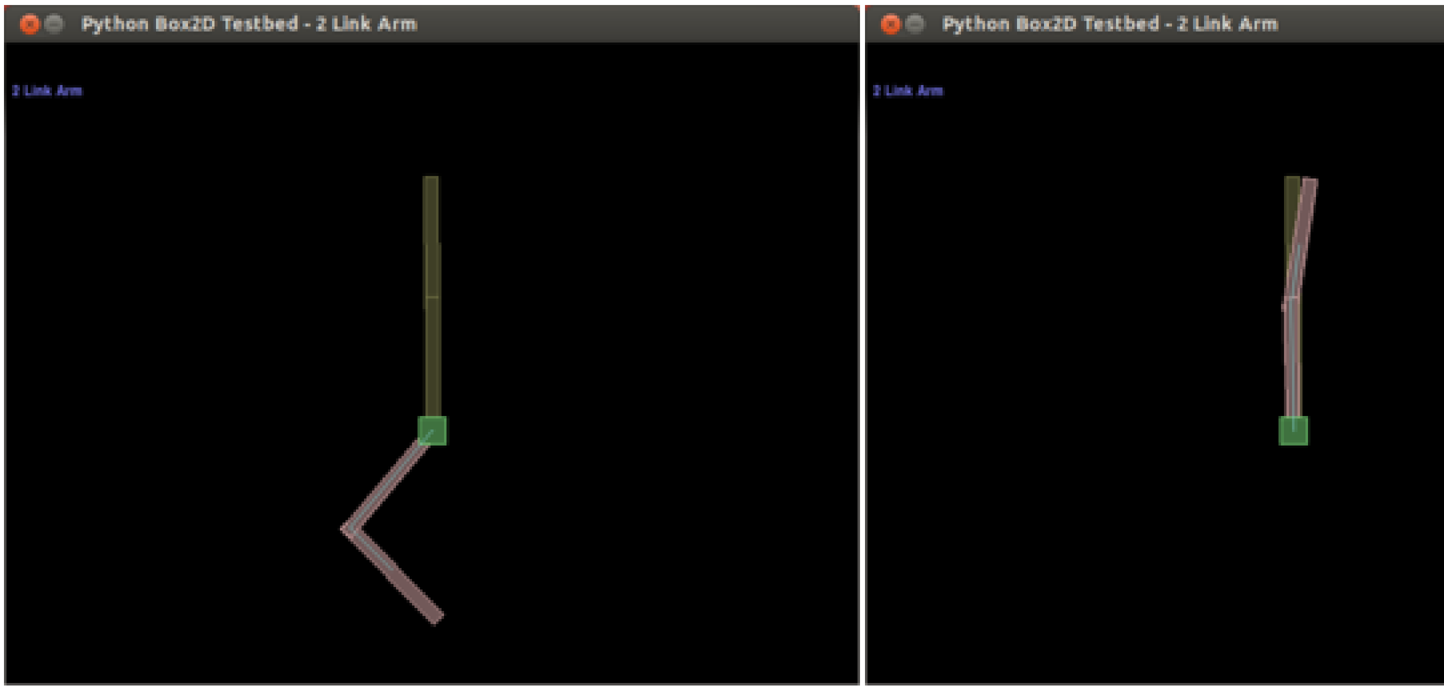


To try it out, run the following from the gps directory:

```
python python/gps/gps_main.py box2d_pointmass_example
```

The progress of the algorithm is displayed on the GUI. The point mass should start reaching the visualized goal by around the 4th iteration.

The second example is a 2-link arm learning to move to goal state.


Python Box2D Testbed - 2 Link Arm
2 Link Arm
Python Box2D Testbed - 2 Link Arm
2 Link Arm

To try it out, run this:

```
python python/gps/gps_main.py box2d_arm_example
```

The arm should start reaching the visualized goal after around 6 iterations.

All settings for these examples are located in `experiments/box2d_[name]_example/hyperparams.py`, which can be modified to input different target positions and change various hyperparameters of the algorihtm.

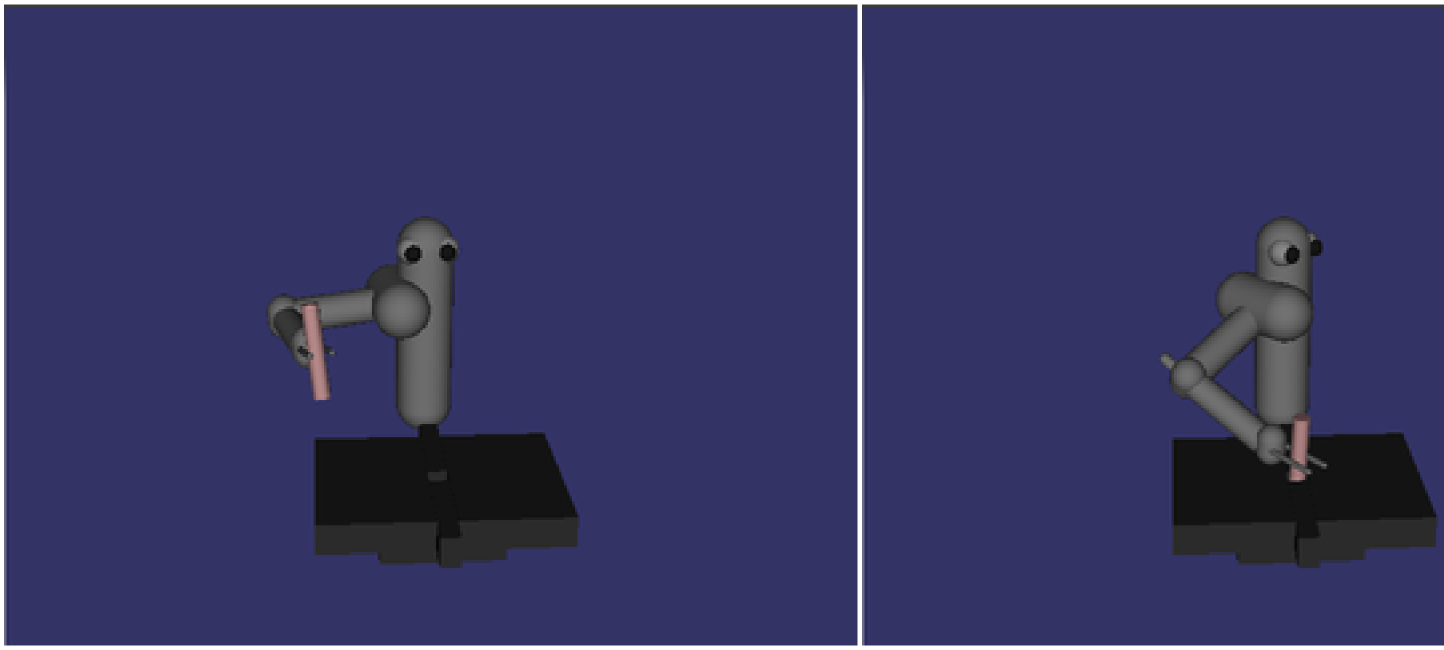
1.3.2 Mujoco example

To run the mujoco example, be sure to first *set up Mujoco*.

The first example is using trajectory optimizing for peg insertion. To try it, run the following from the gps directory:

```
python python/gps/gps_main.py mjc_example
```

Here the robot starts with a random initial controller and learns to insert the peg into the hole. The progress of the algorithm is displayed on the GUI.



Now let's learn to generalize to different positions of the hole. For this, run the guided policy search algorithm:

```
python python/gps/gps_main.py mjc_badmm_example
```

The robot learns a neural network policy for inserting the peg under varying initial conditions.

To tinker with the hyperparameters and input, take a look at `experiments/mjc_badmm_example/hyperparams.py`.

1.3.3 PR2 example

To run the code on a real or simulated PR2, be sure to first follow the instructions above for ROS setup.

1. Start the controller

Real-world PR2

On the PR2 computer, run:

```
roslaunch gps_agent_pkg pr2_real.launch
```

This will stop the default arm controllers and spawn the GPSPR2Plugin.

Simulated PR2

Note: If you are running ROS hydro or later, open the launch file pr2_gazebo_no_controller.launch and change the include line as specified.

Launch gazebo and the GPSPR2Plugin:

```
roslaunch gps_agent_pkg pr2_gazebo.launch
```

2. Run the code

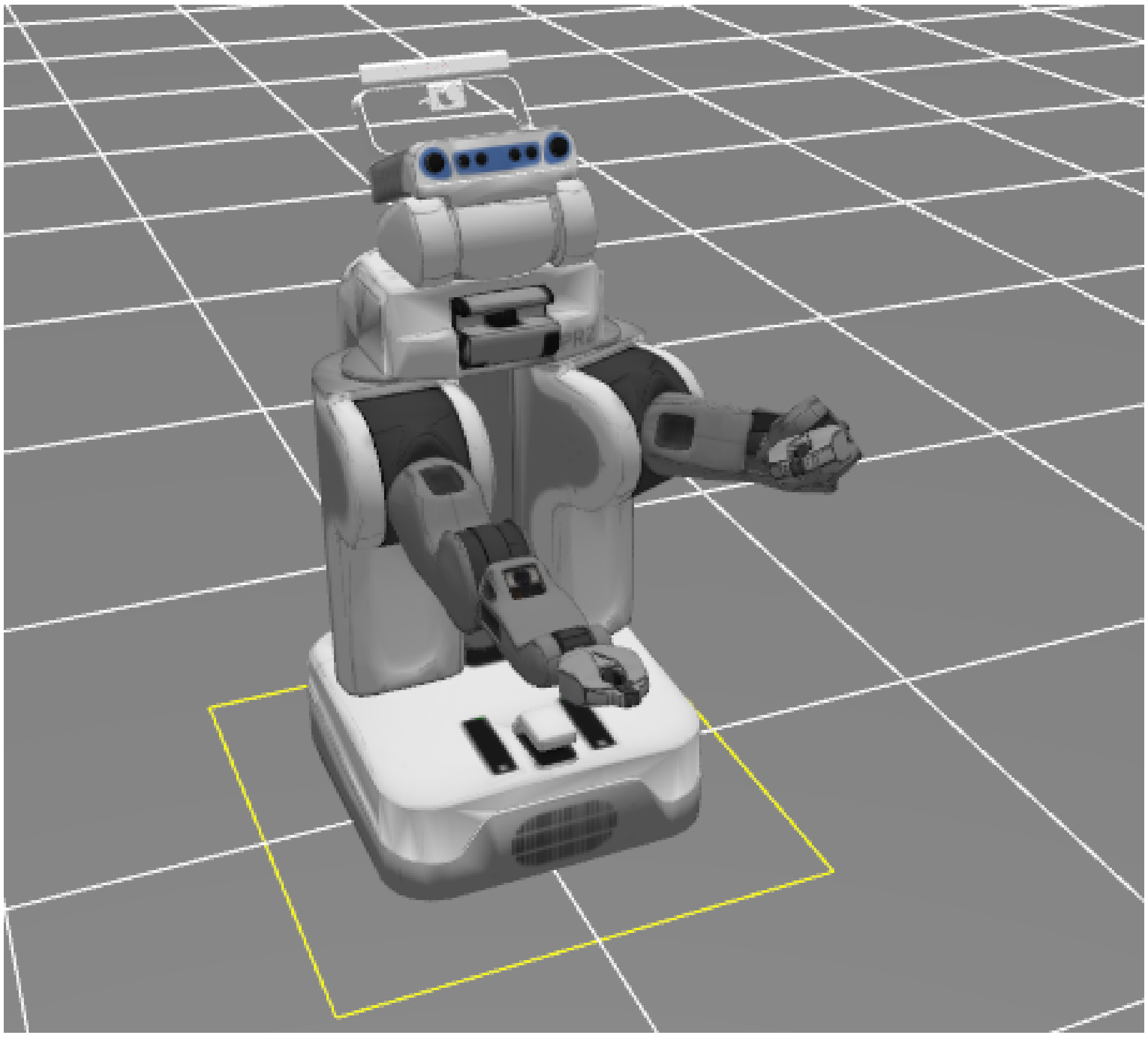
Now you're ready to run the examples via gps_main. This can be done on any machine as long as the ROS environment variables are set appropriately.

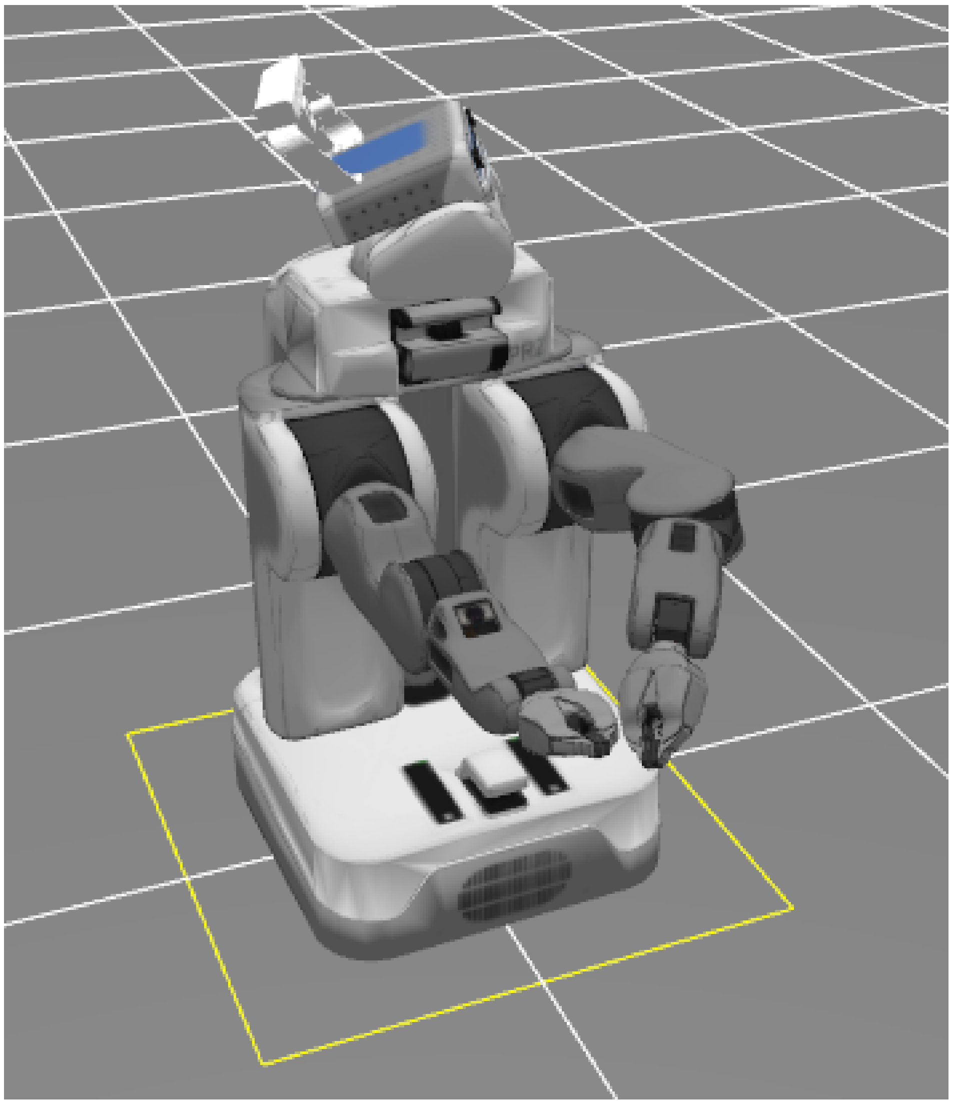
The first example starts from a random initial controller and learns to move the gripper to a specified location.

Run the following from the gps directory:

```
python python/gps/gps_main.py pr2_example
```

The PR2 should reach the position shown on the right below, and reach a cost of around -600 before the end of 10 iterations.


PR2



The second example trains a neural network policy to reach a goal pose from different starting positions, using guided policy search:

```
python python/gps/gps_main.py pr2_badmm_example
```

To learn how to make your own experiment and/or set your own initial and target positions, see *the next section*

1.3.4 Running a new experiment

1. Set up a new experiment directory by running:

   ```
   python python/gps/gps_main.py my_experiment -n
   ```

   This will create a new directory called my_experiment/ in the experiments directory, with a blank hyperparams.py file.

2. Fill in a hyperparams.py file in your experiment. See pr2_example and mjc_example for examples.

3. If you wish to set the initial and/or target positions for the pr2 robot agent, run target setup:

```
python python/gps/gps_main.py my_experiment -t
```

See the GUI documentation for details on using the GUI.

4. Finally, run your experiment

```
python python/gps/gps_main.py my_experiment
```

All of the output logs and data will be routed to your experiment directory. For more details, see intended usage.

1.4 Documentation

In addition to the inline docstrings and comments, see the following pages for more detailed documentation:

- Intended Usage
- GUI for visualization and target setup
- Configuration and Hyperparameters
- FAQ

1.5 Learning with your own robot

The code was written to be modular, to make it easy to hook up your own robot. To do so, either use one of the existing agent interfaces (e.g. AgentROS), or write your own.

1.6 Reporting bugs and getting help

You can post questions on gps-help. If you want to contribute, please post on gps-dev. When your contribution is ready, make a pull request on GitHub.

1.7 Licensing

Intended Usage

The intended command line usage is through the gps_main.py script:

```
cd /path/to/gps
python python/gps/gps_main.py -h
usage: gps_main.py [-h] [-n] [-t] [-r N] experiment

Run the Guided Policy Search algorithm.

positional arguments:
  experiment         experiment name

optional arguments:
  -h, --help         show this help message and exit
  -n, --new          create new experiment
  -t, --targetsetup  run target setup
  -r N, --resume N   resume training from iter N
```

Usage:

- `python python/gps/gps_main.py <EXPERIMENT_NAME> -n`

  Creates a new experiment folder at `experiments/<EXPERIMENT_NAME>` with an empty hyperparams file `hyperparams.py`. Copy and paste your old `hyperparams.py` from your previous experiment and make any modifications.

- `python python/gps/gps_main.py <EXPERIMENT_NAME> -t` (for ROS only)

  Opens the Target Setup GUI, for target setup when using ROS. See the Target Setup GUI section for details.

- `python python/gps/gps_main.py <EXPERIMENT_NAME>`

  Opens the GPS Training GUI and runs the guided policy search algorithm for your specific experiment hyperparams. See the Training GUI section for details.

- `python python/gps/gps_main.py <EXPERIMENT_NAME> -r N`

  Resumes the guided policy search algorithm, loading the algorithm state from iteration N. (The file `experiments/<EXPERIMENT_NAME>/data_files/algorithm_itr_<N>.pkl` must exist.)

For your reference, your experiments folder contains the following:

- `data_files/` - holds the data files.
  - `data_files/algorithm_itr_<N>.pkl` - the algorithm state at iteration N.
  - `data_files/traj_samples_itr_<N>.pkl` - the trajectory samples collected at iteration N.

  - `data_files/pol_samples_itr_<N>.pkl` - the policy samples collected at iteration N.
  - `data_files/figure_itr_<N>.png` - an image of the GPS Training GUI figure at iteration N.
- `hyperparams.py` - the hyperparams used for this experiment. For more details, see this page.
- `log.txt` - the log text of output from the Target Setup GUI and the GPS Training GUI.
- `targets.npz` - the initial and target state used for this experiment (ROS agent only – set for other agents in hyperparams.py)

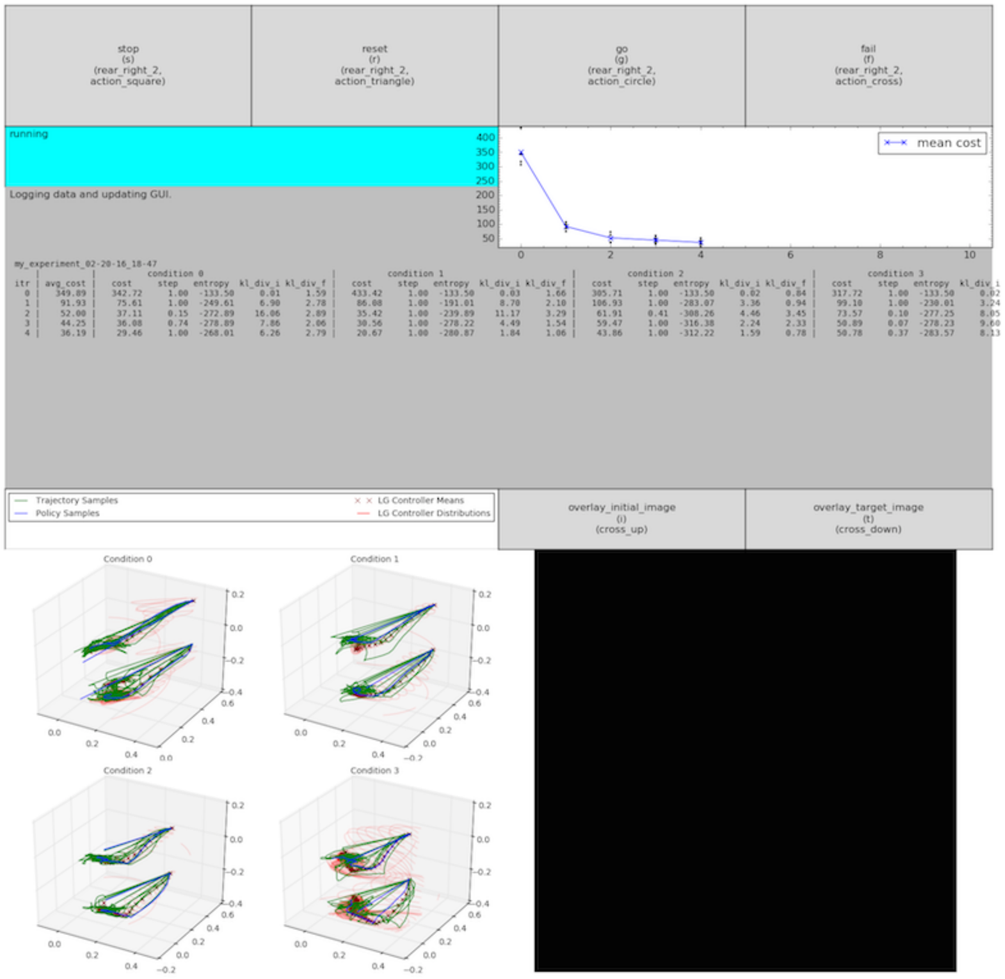
CHAPTER 3

GUI Docs

There are two GUI interfaces which are useful for interacting with and visualizing experiments, documented below.

3.1 GPS Training GUI

```
python python/gps/gps_main.py <EXPERIMENT_NAME>
```



```
running

Logging data and updating GUI.

my_experiment_02-20-16_18-47
    |          |                      condition 0                        |                      condition 1                        |                      condition 2                        |                      condition 3
itr | avg_cost |    cost      step   entropy  kl_div_i kl_div_f |    cost      step   entropy  kl_div_i kl_div_f |    cost      step   entropy  kl_div_i kl_div_f |    cost      step   entropy  kl_div_i
  0 |   349.89 |  342.72      1.00   -133.50      0.01     1.59 |  433.42      1.00   -133.50      0.03     1.66 |  305.71      1.00   -133.50      0.02     0.84 |  317.72      1.00   -133.50      0.02
  1 |    91.93 |   75.61      1.00   -249.61      6.90     2.78 |   86.08      1.00   -191.01      8.70     2.10 |  106.93      1.00   -283.07      3.36     0.94 |   99.10      1.00   -230.01      3.24
  2 |    52.00 |   37.11      0.15   -272.89     16.06     2.89 |   35.42      1.00   -239.89     11.17     3.29 |   61.91      0.41   -308.26      4.46     3.45 |   73.57      0.10   -277.25      8.05
  3 |    44.25 |   36.08      0.74   -278.89      7.86     2.06 |   30.56      1.00   -278.22      4.49     1.54 |   59.47      1.00   -316.38      2.24     2.33 |   50.89      0.07   -278.23      9.60
  4 |    36.19 |   29.46      1.00   -268.01      6.26     2.79 |   20.67      1.00   -280.87      1.84     1.06 |   43.86      1.00   -312.22      1.59     0.78 |   50.78      0.37   -283.57      8.13
```

The GPS Training GUI is composed of seven parts:

The Action Panel: Consists of 4 actions which can be performed by clicking the button, pressing the keyboard shortcut, or using the PS3 Controller shortcut:

- `stop` - stop the robot from collecting samples (after the current sample has completed), used to perform a manual reset of the robot's arms
- `reset` - reset the robot to the initial position (after the current sample has completed), used to perform a manual reset of the objects in the scene
- `go` - start/restart the robot to collect samples (after using `stop`, `reset`, or `fail`), used to resume training after stop, reset or fail
- `fail` - fail the current sample being collected and recollect that sample (after the current sample has completed), used to recollect a sample that was conducted under faulty conditions

The Action Status TextBox: Indicates whether or not the actions were performed successfully or failed.

The Algorithm Status TextBox: Indicates the current status of the algorithm (sampling, calculating, logging data, etc.).

The Cost Plot: After each iteration, plots the cost per condition (as points) and the mean cost (as a connected line between iterations).

The Algorithm Output Textbox: After each iteration, outputs the iteration number and the mean cost, and for each condition, the cost, the step_multiplier, the linear Gaussian controller entropy, and the initial and final KL Divergences (for BADMM only). To change what is printed, see the `update` method of `gps_training_gui.py`.

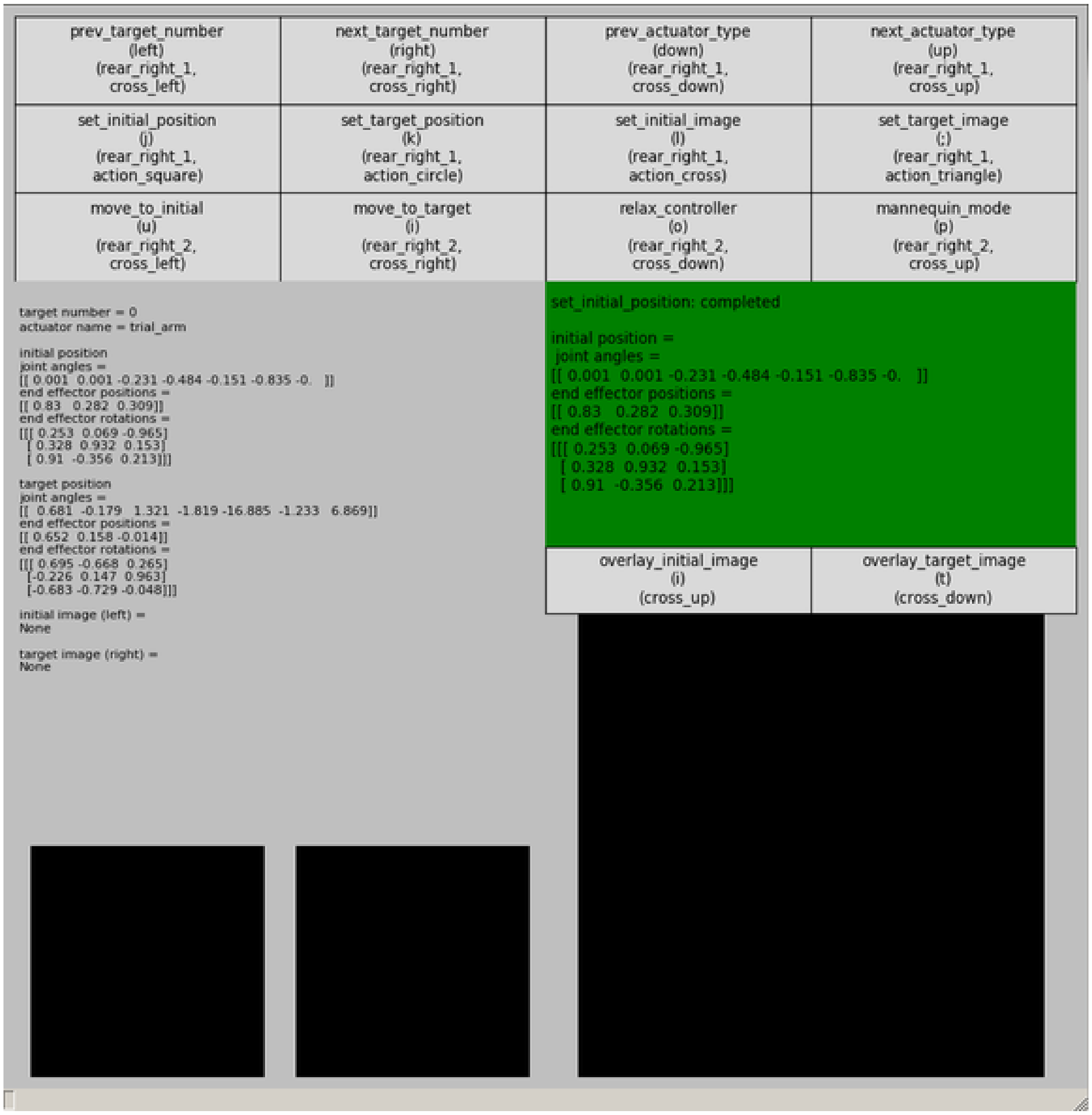
The 3D Trajectory Visualizer: After each iteration, plots the trajectory samples collected (green lines), the policy samples collected (blue lines), the linear gaussian controller means (dark red crosses), and the linear gaussian controller distributions (red ellipses).

The Image Visualizer: Displays the real-time images outputted by the PR2's on-board Kinect. Contains the below two actions:

- `overlay_initial_image` - overlays the initial image set by the Target Setup GUI (press `reset` at the end of sampling to reset the robot to the initial position and check if the image matches the initial image set by the Target Setup GUI
- `overlay_target_image` - overlays the target image set by the Target Setup GUI (press `stop` at the end of sampling to stop the robot at the target position and check if the image matches the target image set by the Target Setup GUI

3.2 Target Setup GUI (for ROS only)

```
python python/gps/gps_main.py <EXPERIMENT_NAME> -t
```


prev_target_number (left) (rear_right_1, cross_left)
next_target_number (right) (rear_right_1, cross_right)
prev_actuator_type (down) (rear_right_1, cross_down)
next_actuator_type (up) (rear_right_1, cross_up)
set_initial_position (j) (rear_right_1, action_square)
set_target_position (k) (rear_right_1, action_circle)
set_initial_image (l) (rear_right_1, action_cross)
set_target_image (;) (rear_right_1, action_triangle)
move_to_initial (u) (rear_right_2, cross_left)
move_to_target (i) (rear_right_2, cross_right)
relax_controller (o) (rear_right_2, cross_down)
mannequin_mode (p) (rear_right_2, cross_up)
overlay_initial_image (i) (cross_up)
overlay_target_image (t) (cross_down)

```
target number = 0
actuator name = trial_arm

initial position
joint angles =
[[ 0.001  0.001 -0.231 -0.484 -0.151 -0.835 -0.   ]]
end effector positions =
[[ 0.83   0.282  0.309]]
end effector rotations =
[[[ 0.253  0.069 -0.965]
  [ 0.328  0.932  0.153]
  [ 0.91  -0.356  0.213]]]

target position
joint angles =
[[  0.681  -0.179   1.321  -1.819 -16.885  -1.233   6.869]]
end effector positions =
[[ 0.652  0.158 -0.014]]
end effector rotations =
[[[ 0.695 -0.668  0.265]
  [-0.226  0.147  0.963]
  [-0.683 -0.729 -0.048]]]

initial image (left) =
None

target image (right) =
None
```

```
set_initial_position: completed

initial position =
 joint angles =
[[ 0.001  0.001 -0.231 -0.484 -0.151 -0.835 -0.   ]]
end effector positions =
[[ 0.83   0.282  0.309]]
end effector rotations =
[[[ 0.253  0.069 -0.965]
  [ 0.328  0.932  0.153]
  [ 0.91  -0.356  0.213]]]
```

The Target Setup GUI is composed of four parts:

The Action Panel: Consists of 12 actions which can be performed by clicking the button, pressing the keyboard shortcut, or using the PS3 Controller shortcut:

- `prev_target_number` - switch to the previous target number (0-9)
- `next_target_number` - switch to the next target number (0-9)
- `next_actuator_type` - switch to the next actuator type (TRIAL_ARM, AUXILIARY_ARM)
- `set_initial_position` - set the initial position (manually move the robot to the desired position and then press set)
- `set_target_position` - set the target position (manually move the robot to the desired position and then press set)

- `set_initial_image` - set the initial image (associated with the initial position)
- `set_target_image` - set the target image (associated with the target position)
- `move_to_initial` - move to the initial position (queries AgentROS to physically move the robot and lock in place)
- `move_to_target` - move to the target position (queries AgentROS to physically move the robot and lock in place)
- `relax_controller` - relaxes the robot controllers, so they can be moved again (after `move_to_initial` or `move_to_target` locks them in place)
- `mannequin_mode` - toggles mannequin mode on or off (for the robot's arms to stay in place after physically moving them)

The Action Status TextBox: Indicates whether or not the actions were performed successfully or failed.

The Targets TextBox: Displays the target values for the current target number and actuator type, including:

- `target number` - the current target number
- `actuator type` - the current actuator type
- `initial pose` - the initial poses' `joint angles`, `end effector points`, and `end effector rotations`
- `target pose` - the target poses' `joint angles`, `end effector points`, and `end effector rotations`
- `initial image` - the initial image associated with the initial pose, displayed on the bottom left
- `target image` - the target image associated with the target pose, displayed on the bottom right

The Image Visualizer: Displays the real-time images outputted by the PR2's on-board Kinect (defaults to the topic /camera/rgb/image_color). Has options to overlay the initial image and the target image.

3.3 GUI FAQ

1. How do I turn off the GPS Training GUI?

At the bottom of the `hyperparams.py` file, in the `config` dictionary set `gui_on` to `False`.

2. Why is the algorithm output text overflowing?

Matplotlib does not handle drawing text very smoothly. Either decrease the amount of iteration data printed (see the `update` function of `gps_training_gui.py`), the size at which is printed (see the `fontsize` parameter in the initialization of the `self._algthm_output OutputAxis` in `gps_trainin_gui.py`), or increase your window size.

3. How do I change the experiment information displayed at the start of learning?

See what algorithm information is printed in the `info` key of the `common` dictionary in `hyperparams.py`.

CHAPTER 4

Configuration & Hyperparameters

All of the configuration settings are stored in a config.py file in the relevant code directory. All hyperparameters can be changed from the default value in the hyperparams.py file for a particular experiment.

This page contains all of the config settings that are exposed via the experiment hyperparams file. See the corresponding config files for more detailed comments on each variable.

4.1 Algorithm and Optimization

Algorithm base class

- initial_state_var
- sample_decrease_var
- kl_step
- init_traj_distr
- sample_increase_var
- cost
- inner_iterations
- dynamics
- min_step_mult
- min_eta
- max_step_mult
- traj_opt

BADMM Algorithm

- fixed_lg_step
- exp_step_decrease
- init_pol_wt
- max_policy_samples
- policy_dual_rate

- inner_iterations
- exp_step_lower
- exp_step_increase
- policy_sample_mode
- exp_step_upper
- lg_step_schedule
- policy_dual_rate_covar
- ent_reg_schedule

LQR Traj Opt

- del0
- eta_error_threshold
- min_eta

Caffe Policy Optimization

- gpu_id
- batch_size
- iterations
- weights_file_prefix
- weight_decay
- init_var
- use_gpu
- ent_reg
- lr
- network_model
- network_arch_params
- lr_policy
- momentum
- solver_type

Policy Prior & GMM

- strength
- keep_samples
- max_clusters
- strength
- min_samples_per_cluster
- max_samples

4.2 Dynamics

Dynamics GMM Prior

- max_clusters
- strength
- min_samples_per_cluster
- max_samples

4.3 Cost Function

State cost

- wp_final_multiplier
- ramp_option
- l2
- data_types
- l1
- alpha

Forward kinematics cost

- evalnorm
- wp_final_multiplier
- alpha
- target_end_effector
- l2
- ramp_option
- env_target
- wp
- l1

Action cost

- wu

Sum of costs

- costs
- weights

4.4 Initialization

Initial Trajectory Distribution - PD initializer

- init_action_offset

- pos_gains
- init_var
- vel_gains_mult

Initial Trajectory Distribution - LQR initializer

- init_acc
- stiffness
- init_gains
- init_var
- stiffness_vel
- final_weight

4.5 Agent Interfaces

Agent base class

- noisy_body_var
- x0var
- dH
- noisy_body_idx
- smooth_noise_renormalize
- smooth_noise
- smooth_noise_var
- pos_body_offset
- pos_body_idx

Box2D agent

Mujoco agent

- substeps

ROS agent

CHAPTER 5

FAQ

What does this codebase include?

This codebase implements the BADMM-based guided policy search algorithm, including LQG-based trajectory optimization, fitting local dynamics model using a Gaussian Mixture Model prior, and the cost functions used our work.

For ease of applying the method, we also provide interfaces with three simulation and robotic platforms: Box2D for its ease of access, Mujoco for its more accurate and capable 3D physics simulation, and ROS for interfacing with real-world robots.

What does this codebase *not* include?

The codebase is a work in progress.

It includes the algorithm detailed in (Levine, Finn et al., 2015) but does not yet include support for images and convolutional networks, which is under development.

It does not include the constrained guided policy search algorithm in (Levine et al., 2015; Levine and Abbeel, 2014). For a discussion on the differences between the BADMM and constrained versions of the algorithm, see (Levine, Finn et al. 2015).

Other extensions on the algorithm, including (Finn et al., 2016; Zhang et al., 2016; Fu et al., 2015) are not currently implemented, but the former two extensions are in progress.

Why Caffe?

Caffe provides a straight-forward interface in python that makes it easy to define, train, and deploy simple architectures.

We plan to add support for TensorFlow to enable training more flexible network architectures.

How can I find more details on the algorithms implemented in this codebase?

For the most complete, up-to-date reference, we recommend this paper:

Sergey Levine*, Chelsea Finn*, Trevor Darrell, Pieter Abbeel. *End-to-End Training of Deep Visuomotor Policies*. 2015. arxiv 1504.00702. [pdf]