Reinforcement Learning with Gaussian Processes for Unmanned Aerial Vehicle Navigation

Nahush Gondhalekar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Pratap Tokekar, Chair
Haibo Zeng
A. Lynn Abbott

June 23, 2017
Blacksburg, Virginia

Keywords: Reinforcement Learning, Gaussian Processes, Unmanned Aerial Vehicle Navigation

Reinforcement Learning with Gaussian Processes for Unmanned Aerial Vehicle Navigation

Nahush Gondhalekar

We present the problem of studying reinforcement learning (RL) for unmanned aerial vehicle (UAV) navigation with few real world samples as possible. A naive implementation suffers from curse of dimensionality in large continuous state spaces. Gaussian Processes(GPs) exploit the spatial correlation to approximate state-action transition dynamics in large state spaces. By incorporating GPs in naive Q-learning we achieve better performance in lesser number of samples. The illustrations are performed using simulations with an aerial robot. Further, we present a Multi-Fidelity Reinforcement Learning (MFRL) algorithm that leverages Gaussian Processes (GP) to learn the optimal policy in a real world environment.In MFRL framework, an agent uses multiple simulators of the real environment to perform actions. With multiple levels of fidelity in a simulator chain, the number of samples used in successively higher simulators can be reduced. This is illustrated with the help of simulations. One of the promising yet fairly unexplored applications of learning autonomous navigation for aerial robots is in Structural Inspection.

This work has been funded by the Center for Unmanned Aircraft Systems (C-UAS), a National Science Foundation-sponsored industry/university cooperative research center (I/UCRC) under NSF Award No. IIP-1161036 along with significant contributions from C-UAS industry members.

Dedication

To Dad and Maa.

Acknowledgments

First and foremost, I would like to express my gratitude towards my advisor, Dr. Pratap Tokekar. Thank you very much for the constant support and motivation. This wouldn't be possible without your encouragement. My sincere acknowledgment for believing in me. I am looking forward to have a continued association and keep learning new things and hoping to collaborate in the future. I could not have asked for a better advisor.

A sincere thank you to my co-author Varun Suryan for his contribution to this thesis. I am thankful to the National Science Foundation for the financial support for the grants enlist the grant #. A special thanks to Dr. Matt Hebdon and his team for the collaboration in the bridge inspection project which is the work presented in chapter **??**.

Robotics Algorithms and Autonomous Systems lab (`http://www.raas.ece.vt.edu/`) has been home for over 18 months; thanks to all the team members who made this time enjoyable. Thank you Ashish Budhiraja, Kevin Yu, Aravind, Lifeng Zhou, Yoonchang Sung, Varun Suryan and Zhongshun Zhang for all the help and fun times at the outdoor experiments. It has been a wonderful one and a half years and I am very proud to be a part of this research group.

I would be short of words to fully express my thanks towards one person who has been an essential part of my life at Virginia Tech. Thank you so much Harsh Patel who has been more of a brother than a friend and has been with me through all the ups and downs and is more than a family to me. Thanks to all the friends in Virginia Tech for the fun times. Thank you Shefali Gundecha who has always been there to support me despite being in a different timezone. Thanks for being there when I needed it the most.

I'd like to express my sincere thanks to the Mccomas Gym and the War Memorial Gym at Virginia Tech which have been my meditation centers and have helped me through the stressful times.

Last but certainly not the least, I cannot describe how thankful I am towards my family. My mother's confidence in me and her endless support has kept me going for these two years. Thank you to my mother Sudha Gondhalekar for being just a phone call away. A special thanks to my dad Ramesh Gondhalekar who's been watching over me from heaven for the past 5 years and I know he would continue to do so.

Contents

List of Figures

Chapter 1

Introduction

With increasing popularity of mobile robots and challenges in their autonomous navigation, a remarkable variety of problems can be phrased as the ones of *Reinforcement Learning* (RL). Unlike the traditional setting of *Perception*, *planning* and *learning* as separate modules, the agent is layered in *task-achieving* modules [**?**]. Each module may be referred to as a *skill* such as "obstacle avoidance" or "following the wall." Programming robots can be a long, time consuming process. A major disadvantage with the behavior-based robots is the laborious programming efforts needed by the designer [**?**] [**?**]. The idea of having *learning* involved is to specify *what* needs to be achieved and let the robot explore the possibilities in order to Figure out *how* to achieve that.

Reinforcement learning in the robotics domain differs considerably from most well-studied reinforcement learning benchmark problems. Problems in robotics are often high-dimensional and are best represented by a continuous state-action space. The true state is not always observable and noise-free. Sometimes, greatly different states may look very similar. It is often essential to know the state of the robot in the environment which gives information with uncertainty. It can be difficult to obtain real-world experience since it is often tedious to obtain and hard to reproduce. In order to learn a particular task at hand within reasonable time, approximations of the environment and system dynamics are used through simulations. However, no matter how costly the real-world experience is, it wholly cannot be replaced by simulations. Smallest of the modeling errors can accumulate to cause a different behavior in the real world.

In particular, obtaining negative samples may require the robot to collide or fail which is undesirable. Hence, it is desirable to minimize the number of real world samples required for learning optimal paths or desired tasks. With a better understanding of reinforcement learning framework for robotics, we may be able to answer fundamental questions such as: What skills are needed to be learned by the robot achieve a particular task? How can we reduce the number of real world samples by optimally building the simulators? How to use approximations in order to learn the desired skill within an acceptable time?.

In this thesis we study these questions for the specific application of Unmanned Aerial Vehicle (UAV). Before introducing the particular problems, we will discuss the challenges in navigation for specific motivating applications.

1.1 Motivation and applications to general scenarios

This work is motivated by the growing interest in using small UAVs for infrastructure and environmental monitoring. UAVs equipped high resolution cameras are being used for bridge inspection [**?**], penstock inspection [**?**], and yield estimation in farms [**?**]. In order for these systems to be successfully deployed, it is crucial that UAVs are able to plan and navigate autonomously in narrow, confined spaces while withstanding large wind disturbances. A controller that is not robust to these disturbances may cause catastrophic failures. Designing custom controllers for UAVs in every new situation is a tedious task and not practical as it requires constant human supervision [**?**]. Instead, reinforcement learning (RL) can provide a general solution.

Navigating safely through a previously-unseen environment is of paramount importance as colliding with obstacles can be catastrophic for a flying object. As shown in Figure **??**, the quadrotor needs to navigate through the narrow corridor flying close to the surfaces for visual inspection. The main challenge here is to design the control robust enough to perform such tasks when the external localization signals are not available. The next section discusses a few of the main challenges in performing structural inspection using UAVs.

Figure 1.1: Quadrotor navigating through a confined space without GPS signal. Images collected during field trial reported in chapter **??**

1.1.1 Challenges

When a UAV needs to inspect a bridge from inside or when it tries to fly near the structures or around the bridge pillars, GPS signal is either not available or not reliable. In order to

be able to autonomously navigate such areas, the UAV must be equipped with an ability to fly close to the surfaces using on-board sensors such as laser range-finders.

A flight control system with conventional methods where the designer needs to be aware of the dynamics of the vehicle and environment increases the development time. If some modifications are made to the vehicle later, most of the controller tuning may be needed to be redone from scratch. Reinforcement learning algorithms provide a number of characteristics which could partly address dynamic environments and dynamics changes: RL can control an agent when a model of the environment is unavailable; RL can control an agent without an existing programmed system; and RL can compensate for changes in the environment as they are encountered since the agent continuously *explores* and *learns*.

One of several immediate questions is: how does the *learning* occur? In many control systems, stability is critically important. Having a *trial and error* way of experimentation may result in a crash which is unacceptable. Trying out experiments in real world is expensive. Development of simulators in order to maximize the performance minimizing the real world costs is crucial. The main contributions of this thesis in development of some RL algorithms centered around UAV navigation reducing real world samples is discussed next.

1.2 Contributions

A Multi-Fidelity Reinforcement Learning (MFRL) algorithm that leverages Gaussian Processes (GPs) to learn the optimal policy in a real world environment is presented in this thesis. In the MFRL framework, an agent uses multiple simulators of the real environment to perform actions. With multiple levels of fidelity in a simulator chain, the number of samples used while learning in successively higher simulators can be reduced. The main contribution of this thesis is to show how we can use GP regression with MFRL to approximate the state-action transition dynamics in large state spaces. GPs exploit the spatial correlation in the transition function for agents that are physical grounded (e.g., a robot). By incorporating GP in the MFRL framework, we achieve further reduction in the number of samples used as we move up the simulator chain. We examine the performance with the help of simulations for navigation with an aerial robot.

We also present an implementation of an *End to End GPQ algorithm* based on a recent work on batch off policy RL with a GP [**?**]. The feedback from the laser sensors is utilized by the learning algorithm in order to *learn* a particular *skill*. The algorithm uses GP regression to learn the Q values associated with the skill of obstacle avoidance in a dynamic environment. The algorithm is testes in a python simulator with a simple kinematic robot model. The learned Q values are transferred to the Gazebo Robot Simulator and an actual quadrotor as the highest fidelity simulator.

1.3 Organization of the Thesis

The thesis is organized in five chapters.

In Chapter **??**, we present a review of the fundamental concepts and background of sequential decision making, reinforcement learning and Gaussian processes.

In Chapter **??**, we present the End-to-End GPQ algorithm which forms the basis of the Multi-Fidelity Reinforcement Learning presented in the next chapter.

Chapter **??** covers Multi-Fidelity Reinforcement Learning with Gaussian Processes. This framework helps to reduce the number of real world samples.

Chapter **??** discusses the motivating application of bridge inspection for the algorithms presented in the thesis along with preliminary field experiments.

We conclude the thesis with an overview of the main contributions and discussions of future work. All the software corresponding to this thesis is available online on github.

Chapter 2

Background

2.1 Sequential Decision Making

A delivery truck trying to decide which house to go first on a tour to deliver the packages, a sudden price drop in the merchandise trying to increase the sales, a robot trying to explore unknown environments. These are all examples of sequential decision making. One of the main factors in sequential decision making is that the decisions made now can have both immediate and long term effects [**?**]. Sometimes it's effective to make a greedy choice but sometimes the decisions made depend critically on future situations. So how to approach sequential decision making?

2.1.1 Approaches to solve sequential decision making problems

Though in this thesis we are mainly interested in the algorithms which deal with *learning*, there exist other algorithms which are used to solve the problems related to sequential decision making. The following ways can be used in approaching sequential decision problems [**?**].

- *Programming*: For each possible event/outcome, try to specify an appropriate or optimal action *a priori*. In most of the general scenarios, this is not possible due to the massive state space of the problem or the intrinsic uncertainty of the environment or both. These solutions may work only for completely known static environments with fixed probability distribution.
- *Search and Planning*: If we know the dynamics of a system, it is easy to plan *Search and Planning* from a defined start state to a goal state. With an uncertainty element added to the environment for the outcomes of the actions, the standard algorithms

would not work. Additionally, we are looking for a general policy for all states of an environment.

- *Learning*: Imagine a robot trying to navigate an unknown maze trying to get out of it. The robot is mounted with an onboard laser sensor for obstacle detection. It's much easier and faster for the robot to interact with the environment to gather data and find the door which gets it to the exit of the maze. So why is *Learning* an effective way to solve the Sequential Decision Problems. The section **??** briefly enlists the advantages.

2.1.2 Why learning?

- No need to perform the tedious task of trying to program all the possibilities in the design phase.
- Learning can effectively cope with uncertain environments, changing states and actions and reward oriented goal finding.
- It can successfully solve the given problem for all the states and come up with a general policy.

2.1.3 Online vs off-line learning

Let's again take an example of a UAV which needs to fly close to a surface of a bridge and take pictures. It is difficult to control it manually with an erratic GPS signal and wind disturbances near the bridge surface. What if the UAV is equipped with a knowledge of maintaining a certain distance from the bridge surface and fly near the bridge autonomously?

This demands for a situation where the UAV needs to be trained to perform the task of flying close to a surface without hitting the obstacle. Learning the controller directly on the real task *Online* is often difficult since learning a task needs a lot of data which sometimes is too time consuming. More importantly, it is not very economic and *safe*, since there is a chance of the quad-rotor colliding several times with the bridge causing financial and other setbacks. It is often desirable to train the robot in a simulator which provides much faster and *safe* training situations where the agent can explore and can afford to make mistakes. *Off-line* learning uses a simulator of the environment as a *cheap* way to gather samples in a *fast* and a *safe* way. Often times one can use the simulators to obtain a reasonable policy for a given problem and then *fine tune* it in the real world.

2.1.4 Rewards, and how to assign them?

The important aspect of Sequential Decision Making is the fact that, if the action is going to result in *good* or *bad* outcomes, cannot be decided right away. Sometimes the first action may have a large influence in reaching the goal even though the actions between the first one and the reward obtained at the end, may be *bad*. A formal model to represent such problems would be extremely useful in analyzing and solving sequential decision making problems. In section **??** we would take a look at the most popular way to represent sequential decision making in an uncertain environment.

2.2 Markov Decision Processes

Markov Decision Processes are popularly used to model sequential decision making when the outcomes of the actions are uncertain. [**?**]. In fact MDPs have become the *de facto* standard formalism for learning sequential decision making. When an action is taken in a particular state, a reward is generated and a next state is attained through a particular transition probability.

Definition 1 *A Markov decision process is a tuple* (S, A, T, R) *in which* S *is a finite set of states,* A *a finite set of actions,* T *a transition function defined as* $T : S \times A \times S \rightarrow [0, 1]$ *and* R *a reward function defined as* $R : S \times A \times S \rightarrow R$.

As seen in Definition **??**, MDPs consist of states, actions, transitions between states and a reward function.

- States: The set of states S is defined as the finite set $\{s^1, ..., s^N\}$ where the size of the state space is N, *i.e.*$|S| = N$. For example, for a robot moving in a $2D$ grid-world like environment, each position (x, y) may be represented as a unique state.
- Actions: The set of actions A is defined as the finite set $\{a^1, ..., a^M\}$ where the size of the action space is M, *i.e.*$|A| = M$. Actions can be used to control the system state. The set of actions that can be applied in some particular state $s \in S$, is denoted $A(s)$, where $A(s) \subseteq A$. For examples, moving the robot forward, backward or sideways can be recognized as actions.
- The Transition Function: When an action $a \in A$ is applied in a state $s \in S$, then the agent makes transition to a state $s' \in S$. This transition is based on a probability distribution over all the states. The probability of ending in state s' when an action a is taken in s, is denoted by $T(s, a, s') \leq 1$.

- The Reward Function: The reward functions determines the reward obtained by the agent for being in a state or performing some action in a state, $R(s, a, s')$ defines the reward obtained by the agent for performing action a in state s and it lands in state s'.

2.2.1 Policies

A policy for an MDP is a function which gives an action $a \in A$ as an output for each state $s \in S$. Formally, a deterministic policy π is a mapping from $\pi : S \rightarrow A$. Optimal policy π^* is computed and used by the agent to try to optimize its behavior in the environment modeled as an MDP. *Optimality* is dependent on what exactly is being optimized? What is the goal of the agent?

2.2.2 Value function and Bellman equation

Most of the learning algorithms for MDPs compute optimal policies by learning value functions. A *value function* represents an estimate of *how good* it is for the agent to be in a certain state (or how good it is to perform a certain action in that state). The notion of *how good* is expressed in terms of an optimality criterion,*i.e.*in terms of the expected return.

The *value function* of a state s under policy π denoted by $V^\pi(s)$ is the expected return when we start in the state s and follow the policy π after that. In an infinite horizon discounted model with discount factor γ, it is expressed as,

$$V^\pi(s) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s \right\} \tag{2.1}$$

A similar state-action value function $Q : S \times A \rightarrow R$ can be expressed as an expected return starting at state s, taking action a and following the policy π.

$$Q^\pi(s, a) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a \right\} \tag{2.2}$$

A very important property of the value function equation is that they satisfy certain recursive relations. For any policy π and state s, the equation **??** can be recursively defined in terms of *Bellman Equation* [**?**].

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s') \Big(R(s, a, s') + \gamma V^\pi(s') \Big) \tag{2.3}$$

Equation **??** represents the expected value of a state s in terms of the immediate reward $R(s, a, s')$ and the value of possible next states $V^{\pi}(s')$ weighted by the transition probabilities $T(s, a, s')$ with a discount factor of γ.

The goal is to find the best policy *i.e.*the policy that maximizes the reward. An *optimal policy* denoted by π^* is such that $V^{\pi^*}(s) \geq V^{\pi}(s)$ for all $s \in S$ and all policies π. The optimal solution can be given by the *Bellman optimality equation* given in equation **??**

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s') \Big(R(s, a, s') + \gamma V^*(s') \Big) \tag{2.4}$$

Similarly, we can express the optimal state-action equation as,

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \Big(R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \Big) \tag{2.5}$$

In case of model-free approaches (section **??**) where T and R are unknown, the Q functions are learned instead of V functions. We can see the relation in equation **??**.

$$V^*(s) = \max_{a} Q^*(s, a) \tag{2.6}$$

So conclusively, to find an optimal policy in case of model-free learning algorithms,

$$\pi^*(s) = arg \max_{a} Q^*(s, a) \tag{2.7}$$

the best action a in a particular state s is the one which has the highest utility based on the possible next state. We call this policy the greedy policy, denoted $\pi_greedy(V)$ because it greedily selects the best action using the value function V.

2.2.3 Solving MDPs

As discussed in the section **??**, finding an optimal policy π^* is an important task of the agent. *Solving* an MDP is nothing but finding the optimal policy π^*. When the model of the MDP is known, the algorithms may be classified under Model Based RL (section **??**) algorithms. Here, we know the transition function and the reward function of the environment. These can be used to find out the value functions and policies using Bellman Equation.(Refer to equation **??**).

In Model-Free RL (section **??**) algorithms, agent relies on the *interaction* with the environment. *Simulations* or actual execution of the policy leads to collect *samples* of transition functions and rewards which are thereby used to determine the state-action value functions.

To directly compute the value functions for the state-action pairs when the model of the environment is not available or to determine the model by interacting with the environment in order to maximize the performance, the agent needs to posses artificial intelligence. We will discuss this branch of machine learning the next section (section **??**).

2.3 Reinforcement Learning (RL)

Reinforcement Learning is a class of problems where the agent needs to learn the *behavior* with trial and error by interacting with a dynamic environment. The two main methods to approach Reinforcement Learning Problems are:

- To search the entire behavior space in order to find the optimal one for the given environment. Examples of these kind of algorithms are genetic programming algorithms.
- Use of statistical methods and dynamic programming to estimate the effect of actions in the states of the given environment. [**?**]

2.3.1 Model of Reinforcement Learning

A simplified Reinforcement Learning Model can be represented as shown in Figure **??** For-

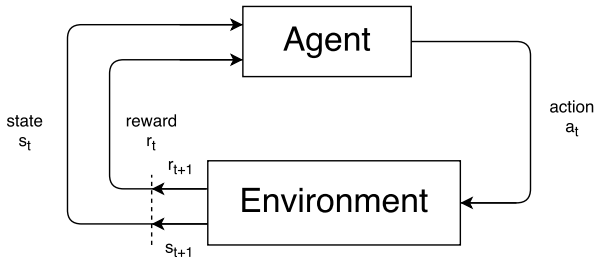


Figure 2.1: A Simplified Reinforcement Learning Model

mally, the model consists of a set of environment states, S a set of agent actions, A; and a set of scalar reinforcement signals which are typically known as rewards. The agent's job is to find a policy π, mapping states to actions ($\pi : S \rightarrow A$) to maximize some long term measure known as *reward*. The environment is typically assumed to be stochastic *i.e.*the same action in the same state at two different occasions may result in two different next states and/or two different rewards. The main difference between Reinforcement Learning and Supervised Learning is that, in RL, the agent needs to gather experience by interacting with the environment to find out the system states, transitions, actions and rewards whereas in Supervised Learning, some information about prior input/output pairs is used to approximate the mapping function.

2.3.2 Exploration vs. Exploitation

An RL agent needs to explicitly explore the environment. One of the classic problems in the literature known as the *k-armed bandit problem* [**?**] may best illustrate the concept of exploration Vs. exploitation. The agent is in a room with a collection of k gambling machines. The agent is permitted a fixed number of pulls, z. Any arm may be pulled on each turn. The machines do not require a deposit to play; the only cost is in wasting a pull playing a suboptimal machine. When arm i is pulled, machine i pays 1 or 0, according to some underlying probability parameter p_i, where the pays are independent events and the p_is are unknown. What should the agent's strategy be to maximize the payoffs?

The agent may believe that some arm may have a higher payoff probability. The question now is, whether to choose the same arm all the time or to explore the other arm that has less information but seems to be better? Depending on how long the game is going to last, if the game is going to last longer, then the agent should not converge to a sub-optimal arm too early and instead explore more. In short, exploitation means to use the knowledge that the agent has found for the current state s by doing one of the actions a that maximizes the value function of the state whereas exploration means, in order to build a better estimate of the optimal value function it should select a different action from the one that it currently thinks is best.

We will see one of the ways to trade of exploration-exploitation using ϵ-greedy strategy used in Q-learning which is a form of model-free learning. The section **??** describes the two fundamental types of Reinforcement Learning.

2.3.3 Classification/Types

As we have seen previously, Reinforcement Learning agent needs to know the model of the environment; The transition function $T(s, a, s')$ and the reward function $R(s, a)$ in order to find an optimal policy. How to find an algorithm to find an optimal policy when such a model is not known in advance? There may be two approaches we can take [**?**].

- Model-free: Learn a controller without learning the model.
- Model-based: Learn the model of the environment in order to derive the controller.

Model-free

Model-free algorithms are the algorithms which do not have an explicit knowledge of the environment. The evaluation of *how good the actions are* is done through trial and error. The space complexity of such algorithms may be considered to be asymptotically less than the space required to store an MDP [**?**]. Model free algorithms tend to keep track of the

value functions only unlike the model-based algorithms which tend to store the environment model information.

The limitation of model-free algorithms may be stated as; these algorithms need **extensive experience** to find an optimal policy. A few examples of model-free algorithms are; *Q-Learning* (section: **??**) and SARSA (State Action Reward State Action).

Model-based

The on-line algorithms such as Q-Learning guarantee convergence if the approximations are accurate. It is expensive to run these algorithms in the real world. In model-based algorithms, an agent tries to learn the model of the environment while obtaining on-line experience and then this model can be used to facilitate learning.

Given a state s and an action a, the probability of going to the next state s' is given by the transition function $T(s, a, a')$. The reward obtained by taking an action a in state s is given by the reward function $R(s, a)$. $T(s, a, s')$ and $R(s, a)$ can be learned and be used during the further runs in order to improve the convergence. It is expected that, during the subsequent runs of the algorithm, the model should already be sufficiently learned to speed up convergence. Extensive amount of computation is required in model-based algorithms. A few examples of model-based reinforcement learning algorithms are; *R-MAX* [**?**] and E^3 [**?**].

2.3.4 Q-Learning

What is Q-Learning?

The basic concept behind Q-learning is that we have a representation of the environmental states $s \in S$ and actions $a \in A$, and an agent is trying to learn the value of each of these actions in each of these states. The agent maintains a table $Q[s, a]$ and at each state s tries to choose the action a which maximizes the function $Q(s, a)$. The formal definition of the Q function is given as follows:

$$Q(s, a) = r(s, a) + \gamma \max_{a'}(Q(s', a')) \tag{2.8}$$

where,
$r(s, a)$ is the immediate reward,
γ is the relative value of the delayed Vs. immediate reward, also known as *discount factor*
$\gamma \in (0, 1]$,
s' is the new state after action a is taken in state s,
a, a' are the actions in state s and s' respectively.
The selected action $\pi(s) = \arg\max_a Q(s, a)$.

2.3.5 Q-Learning Algorithm

Input : S is a set of states
A is a set of action
γ is the discount factor
α is the learning rate or step size ($\alpha \in (0, 1)$)
Local data: Store the whole Q matrix $Q[S, A]$,
previous state s
previous action a,
next state s'
Initialize : Initialize $Q[S, A]$ arbitrarily/Zero
while *Termination condition not reached* **do**
  Choose an action a in the current state s based on the current Q-value estimates.
  Perform the action a in the state s and observe the current reward r and the next state s'.
  Update $Q(s, a) := Q(s, a) + \alpha[r(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)]$:
end

Algorithm 1: Q-Learning Algorithm

The update equation given in algorithm **??** updates the Q-value of the last state-action pair (s,a) with respect to the observed next state s' and the reward r with a learning rate parameter $\alpha \in (0, 1)$. Recalling Bellman update equation from section **??**, we can see that updated Q-value is the expected sum of the reward and discounted value of the next state. Unlike a general Bellman update equation, we are not marginalizing over all possible next states, since we have only observed one state here along with the reward for a particular state-action pair.

However, we can control the parameter α, which is the learning rate so as to allow the Q-value to change slightly from its old estimate to a new estimate in the direction of the observed state and reward.

The need of learning Q values.

Consider an agent that has to navigate through a grid given in the Figure **??**. The walls in the grid block the agent's path. The actions taken by the agent are noisy *i.e.* 80% of the time, the action North takes the agent North if there's no wall, 10% of the time action North takes the agent West, and 10% of the times to East. Each step taken by the agent has a small negative reward say -0.25. A reward of $+1$ or -1 is given at the states shown in Figure **??**. The goal of the agent is to maximize the sum of the rewards.

This problem can be solved using value iteration where the Bellman optimality equation (Refer to equation **??**) characterize the optimal values. The agent has to store the Q values

for each state-action pair ($Q(s, a)$). The space required to store the Q-table is $S \times A$. In this example, the number of states are 11 and number of possible actions are 4. The efficiency of recursive Bellman update is $O(S^2)$. Imagine when the state space becomes larger and the time complexity increases with the square of number of states. A naive Q-Learning approach wouldn't be able to scale up to give faster solutions as the problem space grows. We will see that in the following scenario.

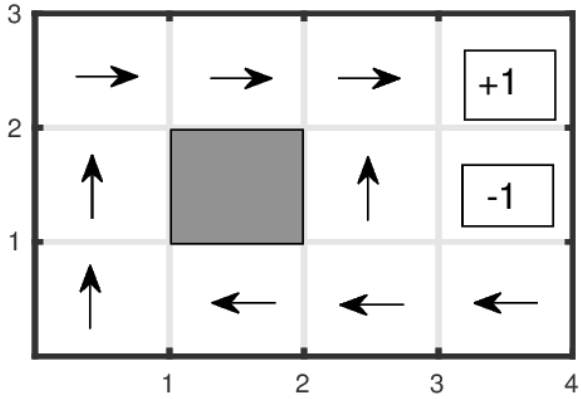


Figure 2.2: An example gridlworld with stochastic actions

Learning Laser data and space complexity!

Consider the following example. A robot (represented in green) with 6 laser sensors is trying to navigate in an environment without colliding with obstacles of different sizes which are moving with different speeds. (The white obstacles are moving at slower speeds whereas the orange obstacle moves faster). The laser sensor can output distance values in the range 0

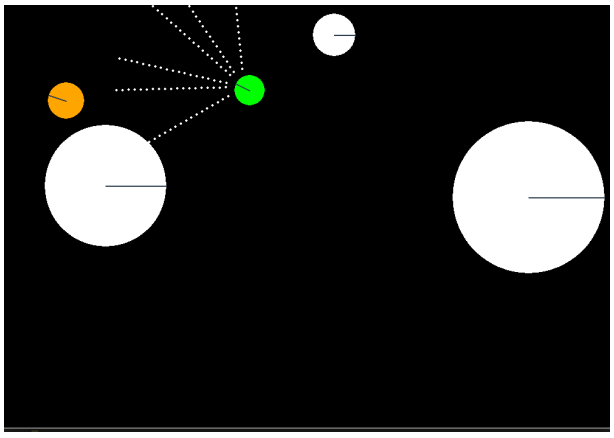


Figure 2.3: Navigation with perception in the loop

to 19. If we consider the state space to be the laser data $[l_1, l_2, l_3, l_4, l_5, l_6]$ and each of the 6 sensors can give 20 values, then we have a total of 2×10^6 possible states. In terms of space and time complexity to learn the $Q[S, A]$ and eventually an optimal policy, this state space is too large for a naive Q-Learning algorithm to achieve faster convergence. As we have seen in section **??**, model-free algorithms need extensive experience in order to compute optimal policies. To experience states of the order of 10^6 is a lengthy process. What could be done to approximate and better estimate all the $Q[S, A]$ values based on the observed set of Q values?

2.4 Introduction to Gaussian Processes (GP)

What are GPs ?

Consider a typical example of a prediction problem given in the Figure **??**. Given some noisy observations at a certain values of the variable x, what is the best estimate at a new value, x^*. We can assume the underlying function $f(x)$ to be some polynomial function and try to select an appropriate regression model to fit the given data. We have numerous possibilities of the function being a quadratic, cubic or even an exponential function. *Gaussian Process Regression* provides a subtle approach in supervised learning in order to learn input-output mappings from empirical data also known as the *training set*.

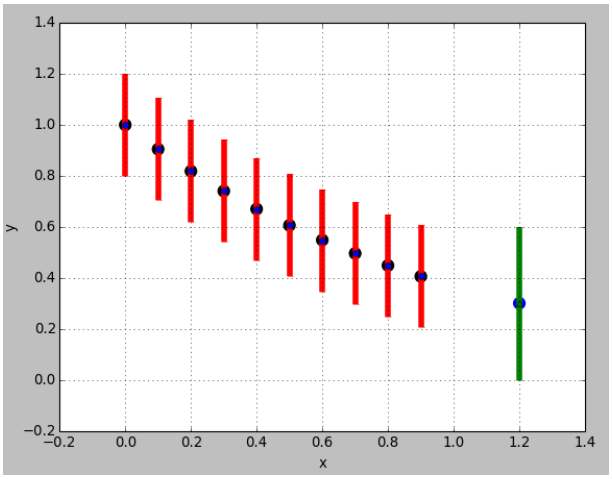


Figure 2.4: Given ten noisy data points (error bars are indicated with vertical lines), we are interested in estimating the value of the eleventh point at $x^* = 1.2$.

A Gaussian Process(GP) is a generalization of the Gaussian Probability Distribution. GPs extend multivariate Gaussian distributions to infinite dimensions. A GP is considered to be a collection of random variables, where any finite subset of which has a joint Gaussian distribution function and a *covariance kernel*. For example, a set of n arbitrary data points y

$= \{y_1, y_2, \ldots\ y_n\}$ can be considered as a single sample from an n-variate Gaussian distribution. It is often assumed that it is a zero mean GP. The relation between the observations is given by the *covariance function/ kernel function*, $k(x, x')$.

Kernel Functions

A few basic kernels are defined below for input points x and x' :

- Squared-Exponential (SE) : This is a popular choice of a kernel function which is also known as the *Radial Basis Function* kernel (RBF), or the *Gaussian* kernel. It has the following form.

$$k(x, x') = {\sigma_f}^2 \exp\left\{\frac{-(x - x')^2}{2l^2}\right\} \tag{2.9}$$

  where the maximum allowable covariance is defined as ${\sigma_f}^2$. $k(x, x')$ approaches this maximum when $f(x)$ is nearly perfectly correlated with $f(x')$.

- Periodic The periodic kernel allows one to model functions which repeat themselves exactly. It has the following form.

$$k_{Per}(x, x') = \sigma^2 \exp\left\{-\frac{2\sin^2(\pi|x - x'|/p)}{l^2}\right\} \tag{2.10}$$

- Linear Using a linear kernel in a GP, is equivalent to doing Bayesian linear regression. It has the following form.

$$k_{Lin}(x, x') = {\sigma_b}^2 + {\sigma_v}^2(x - c)(x' - c) \tag{2.11}$$

  - The offset c determines the x-coordinate of the point that all the lines in the posterior go though. The mean will be zero unless noise is added.
  - The constant variance ${\sigma_b}^2$ determines how far from 0 the height of the function will be at zero. It is putting a prior to the value of the function instead of specifying the value of the function directly.

In all the above kernels,

- **Lengthscale** l : The lengthscale l describes how smooth a function is. Small lengthscale value means that function values can change quickly, large values characterize functions that change only slowly. In general, we won't be able to extrapolate more than l units away from your data.

- **Signal variance** σ^2 : The signal variance σ^2 determines the average distance of your function away from its mean. It describes the variation of function values from their mean.

[?]

Kernel Parameters Each kernel has a number of parameters which are used to specify the precise shape of the covariance function. These are referred to as *hyper-parameters*, since they can be viewed as specifying a distribution over function parameters, instead of being parameters which specify a function directly. An example would be the lengthscale parameter $l$of the RBF kernel, which specifies the width of the kernel and thereby the smoothness of the functions in the model.

Stationary and Non-stationary Kernels The SE and Periodic kernels are *stationary*. This implies that their value only depends on the difference $x \quad x'$. The probability of observing a particular dataset remains the same even if we move all the x values by some offset. In contrast, the linear kernel is *non-stationary*, This means that the corresponding GP model will produce different predictions if the data were moved while the kernel parameters were kept fixed.

2.4.1 Gaussian Process Regression

Consider a least-square line fit $\hat{y} = p_0 + p_1 x$. We are specifying two parameters p_0 and p_1 for the Figure **??**. A better fit would be to use a quadratic function say $\hat{y} = p_0 + p_1 x + p_2 x^2$. In this case, we need 3 parameters p_0, p_1 and p_2. What if we would like to consider all possible functions without specifying the number of parameters? So in GPs, there are no parameters *i.e.*there are infinitely many parameters.

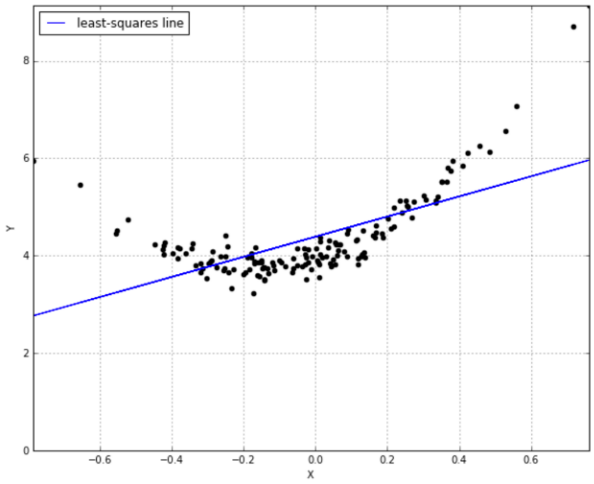


Figure 2.5: To find a function that is consistent over the observed data

The mathematical foundation of GPs is the multivariate Gaussian distribution. Consider the bivariate normal distribution given in Figure **??**. The shape of the bell curve is determined by the covariance matrix. If we imagine a horizontal plane slicing the bell curve and if we see

a circle, that means the two independent variables are normally distributed meaning their covariance is 0.

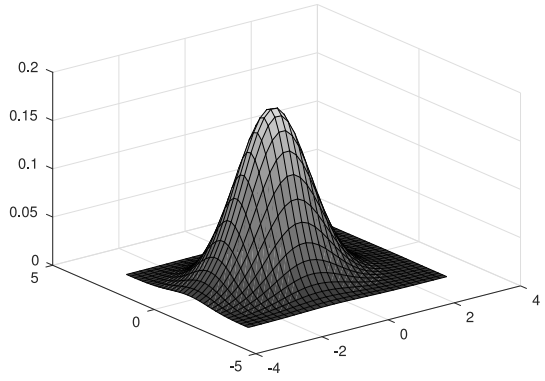


Figure 2.6: A Bivariate Normal Distribution

Given a joint probability distribution of the variables x_1 and x_2 as:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} \sigma_{11} & \sigma_{12} \\ \sigma_{21} & \sigma_{22} \end{pmatrix}\right) \tag{2.12}$$

It is possible to get the conditional probability of $x_1|x_2$ or vice-versa. In a GP, we can derive the posterior from the prior and the observations; it's just that, we are talking about the joint probability all the values of the function $f(x)$ for all the values of x within the specified bounds.

To sum it up, the posterior is the joint probability of the outcome of the values of the function out of which some are observed (denoted by f) and some are not (denoted by f^*).

$$\begin{pmatrix} f \\ {f^*}_2 \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu \\ \mu_* \end{pmatrix}, \begin{pmatrix} K & K_* \\ {K_*}^T & K_{**} \end{pmatrix}\right) \tag{2.13}$$

Here, K is the matrix we get by applying the kernel function to the observed values, *i.e.*the similarity between each observed x .K_* represents the similarity of the training values to the test values whose output values are to be estimated. K_{**} gives the similarity of the test values with each other. We are ultimately trying to determine $p(f_*|x_*, x, f)$ where f and f_* are jointly distributed as shown in equation **??**. Skipping the proof details, we ultimately are able to define the distribution $f_* \sim \mathcal{N}(\mu_*, \Sigma_*)$, where μ_* is the mean and Σ_* is the covariance matrix.

2.4.2 Gaussian Processes for Machine Learning

As stated in [**?**], GPs can be used to model system dynamics. Assuming a set of N-dimensional observations of the form (s, a, s'), we can use separate Gaussian Process model for predicting each coordinate of the system dynamics. For example, the inputs can be the state-action pair (s, a) and the output is a Gaussian distribution over $y = s'_n$, $n = 1, 2, ...N$. A multivariate Gaussian distribution over the next state and transition probability can be obtained. GPs can be used to model value functions as a function of states. We'd typically specify a finite number of data points $S = s_1, s_2, ..s_m$ and let the GP predict it over the space.

The Gaussian processes offer nice properties such as uncertainty estimates over the function values, robustness to over-fitting, and principled ways for hyper-parameter tuning. In general, these useful properties of GPs are exploited in order to make them popular in terms of their use in machine learning.

2.5 Reinforcement Learning for UAVs

The concept of using reinforcement learning techniques for the control of unmanned aerial vehicles is not new; it has peaked interest outside of academia. Most of the existing literature related to using learning algorithms in a UAV context mainly discusses high level planning or trajectory control. More traditional low level control along with the implantation of machine learning algorithms to learn a set of skills or planning the desired trajectories is employed.

[**?**] discusses about learning the non-linear control for advanced stabilization of the UAVs whereas [**?**] developed a more high level control implementing learning strategies for high speed quadrotor multi flips. The system is able to do a series of flips for a quadrotor demonstrating online learning of acrobatics.These are some of the examples of reinforcement learning algorithms being applied to a low level motion control of UAV.

Some of the early examples of robot reinforcement learning are; The OBELIX robot which learned to push boxes [**?**] using value function-based approach, Carnegie Mellons autonomous helicopter leveraged a model-based policy-search approach to learn a robust flight controller [**?**], Petar Kormushev and Sylvain Calinontaught from Italian Institute of Technology taught a robot the technique to flip pancakes [**?**]. The autonomous UAV learns the landing skill from scratch by interacting with the environment. The reinforcement learning algorithm explored in [**?**] is Least-Squares Policy Iteration (LSPI) to gain a fast learning process and a smooth landing trajectory. A recent paper [**?**] applied reinforcement learning to a UAV to achieve stable hovering using the well known technique of Q-learning (section **??**). Unlike most of the studies, experiments using an actual quadrotor UAV were performed and the experimental results demonstrate that the UAV can acquire knowledge to achieve stable hovering over a marker placed on the ground.

However in this study we explore the use of Gaussian Processes on Reinforcement Learning algorithms for UAV navigation.

2.6 Deep Reinforcement Learning

Reinforcement learning is a branch of machine learning primarily based on sequential decision making (refer to section **??**). The conventional reinforcement learning focuses on linear function approximations and tabular representations [**?**]. To generalize in large and continuous state and action spaces, the linear approximations and tabular functions are not enough [**?**]. Performance of most of the successful RL applications operating in high dimensional sensor inputs like vision and speech rely on the quality of features along with the function approximations or policy representation.

[**?**] states that, with sufficient data fed into a deep neural network, it is often possible to learn better representations compared to handcrafted features. [**?**] describes a Deep-RL system which combines Deep Neural Networks with Reinforcement Learning at scale for the first time. This work represents the first demonstration of a general-purpose agent that is able to continually adapt its behavior without any human intervention. The algorithm - *Deep Q-learning with experience replay* for deep Q learning given in [**?**] is advantageous over an online Q learning algorithm in the following ways.

- Each step of experience may be used in many weight updates which results in greater data efficiency.
- Due to strong correlation, learning from consecutive samples may be inefficient. Randomizing samples leads to decrease in covariance of the updates.

.

Basic steps of applying RL to deep neural networks are:

- Use of a deep network to represent value function or policy or model.
- End to end optimization of this value function or policy or model.
- Use of Stochastic Gradient Decent to update the weights.

[**?**] A naive representation of a Deep-Q network is shown in Figure **??**.

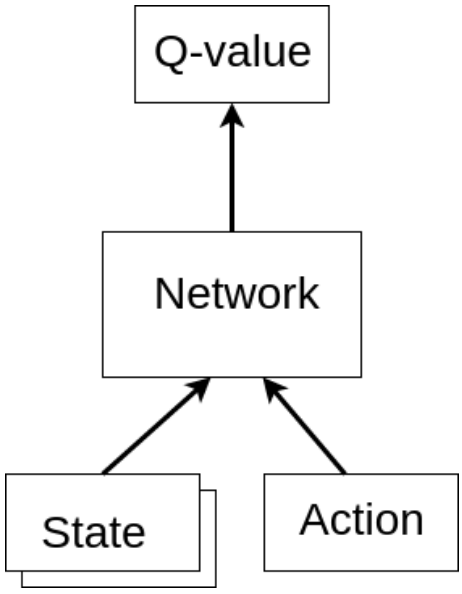


Figure 2.7: A naive representation of deep Q-network

Chapter 3

The End-to-End GPQ Algorithm

3.1 Perception in the loop

Perception is how the robot is aware of the dynamics of the environment. Just as humans perceive things with their ey es, robots use sensors in order to be aware of its surroundings. Perception usually provides data which allows the robot to determine where it is as well as how far the obstacles or goals or adversaries are. The robot must be able to make intelligent decisions based on the data it gathers. Sensors are very critical for intelligent robots to carry out tasks in dynamic environment which limit and define the capability of a robot [**?**]. In order for the robot to be able to navigate through an environment or perform a task of obstacle avoidance it needs to be equipped with sensors which are able to tell what surrounds it.

Perception in the loop may be explained with this example. Imagine a robot equipped with a laser sensor (as shown in Figure **??**) which provides the distance between the robot and its nearest obstacle essentially informing the robot about its surroundings. The task of the robot is to navigate through this environment without crashing into obstacles assuming the robot has no prior information about the environment and only has access to the real time laser data. How can we use the laser data in order to carry out the task?

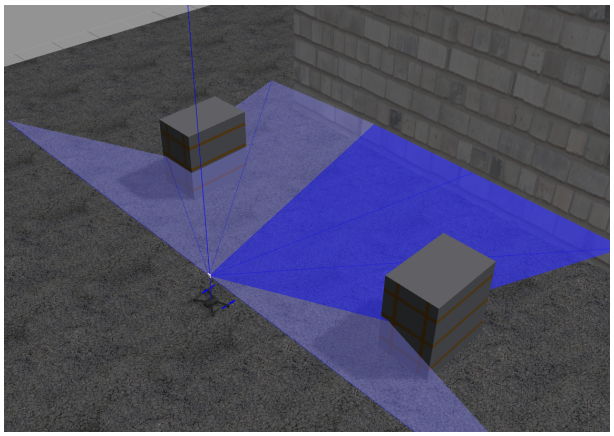


Figure 3.1: A UAV equipped with a laser sensor

3.2 The Simulator setup

caption Two simulators were implemented in order to present the idea of perception and subsequently implementing the idea of reinforcement learning in order to learn a *skill*.

- **Simulator 1**: A python based obstacle avoidance simulator: As shown in Figure **??**, the robot is trying to navigate without colliding with obstacles in a dynamic environment setting. The white obstacles move slower compared to the orange obstacle. The robot is equipped with a laser sensor which uses 6 beams in order to find out the distances from the obstacles at different angles. The robot has a constant speed and can perform 3 actions; turn left, turn right or stay in the same direction.

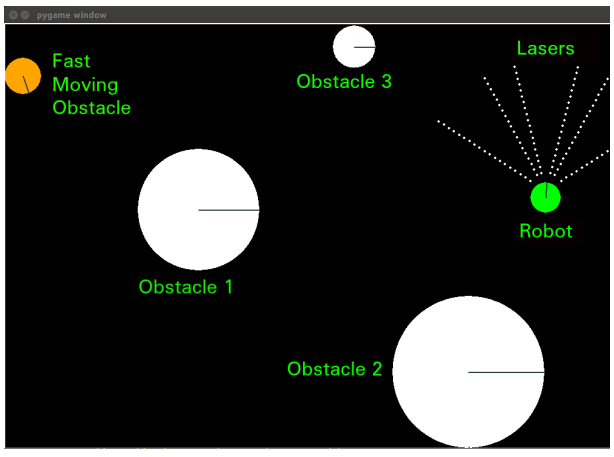


Figure 3.2: A python based simulator for obstacle avoidance

- **Simulator 2**: A second simulator is implemented in the Gazebo robot simulator [**?**]. A UAV equipped with a Hokuyo Laser is implemented in the simulator as shown in Figure **??**. This may be considered as a 3D extension of the simulator shown in Figure **??**. With infinite state-action space and ability to add robot dynamics, this simulator is a higher fidelity simulator compared to the python-pygame based 2D simulator.

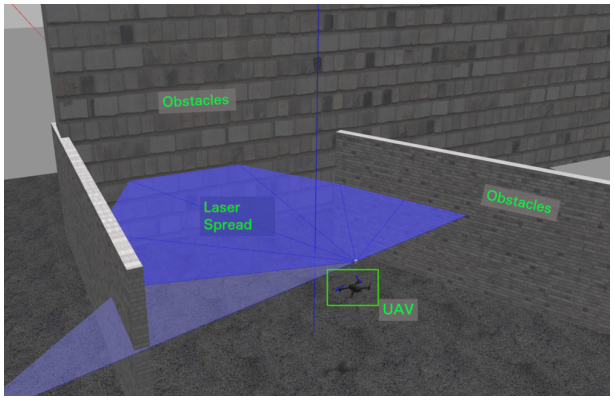


Figure 3.3: A UAV equipped with laser sensor in Gazebo Robot Simulator

In both of the above simulators the main principle of *perception in the loop* is used in order to use the laser data to find out the current position of the obstacle with respect to the robot and take an optimal action in order to avoid the obstacle. In order to learn *obstacle avoidance* as a *skill* the algorithm must be agnostic to the shape or size of the obstacle and it should work in case of moving obstacles as well.

3.2.1 Laser data and Q-Learning

In order to learn the *skill* of obstacle avoidance, we decided to learn the mapping from sensor values to action. A naive Q-Learning algorithm was implemented (refer to algorithm **??**) on the simulators described in section **??**. For the python based simulator, any state $s \in S$ is defined as a set of laser sensor values $\{l_1, l_2, l_3, l_4, l_5, l_6\}$, where $l_i \in \mathcal{P}(\mathbb{Z})$ Action a can be chosen from a set of action $A := \{0, 1, 2\}$ where action $= 0$ turns the robot left by a certain angle, action $= 1$ turns the robot right by a certain angle and action $= 2$ does not change the orientation of the robot. A reward of -500 is given on a crash else the reward is an increasing function of the sum of the readings of the 6 laser sensors. (higher the readings farther the robot is from the obstacle which must be rewarded well.)

In case of the Gazebo Robot Simulator, the state-space S and the action-space A are both continuous spaces. Assuming there are 8 laser beams, the state s may be represented as $\{l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8\}$. Here l_i can take any real value from 0 to 6. (The Hokuyo sensor is simulated in Gazebo Simulator with a maximum range of 6 meters.) The action a may be

represented as a 3D vector $x_{\hat{i}} + y_{\hat{j}} + z_{\hat{k}}$. By excluding the robot dynamic constraints from the analysis, we have infinite possible values for a state-action pair (s, a).

Let us consider a static environment first. Assuming each sensor can give 20 discrete values in case of Simulator 1 given in section **??**, we have about 2 *million* possible states. To observe all of the states in order to be able to store and update the value function (or Q values) is a tedious task. Once a sufficient number of states are observed for the optimal Q values, the robot may learn to navigate through the environment successfully. This may take a longer time due to the size of the state space. In case of simulator 2, the state space is infinite and hence it is not possible to observe all the possible state-action pairs.

3.3 End-to-End GPQ Algorithm

A Gaussian Process model of the value function (Q) is presented in [**?**] in both batch and online settings. Consider the problem of finding the value function for over a *million* state-action pairs in limited time described in section **??**. The problem here closely relates to the problem in section **??** where we were tasked to find out a function to fit the observed data. Given a set of observed state-action pairs (s, a) and the reward r, we calculate the Q values for the observed state-action pairs (s, a). Using Gaussian Process Regression (section **??**), we predict the Q values over the entire state space S (Refer algorithm **??**).

The batch off-line algorithm may be considered in two phases.

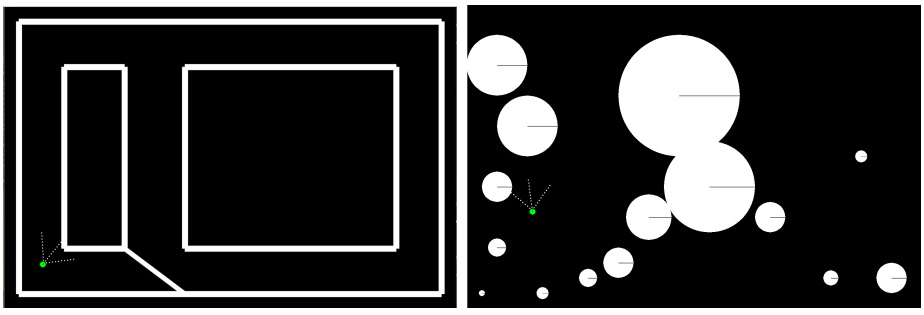
- Training phase: Refer to the lines 1 to 9 in algorithm **??**. Take a random action a in state s. Let the observed state and the reward be s' and r respectively. The new data (s, a, r, s') is added to the training set T. This loop is executed for a predefined number of iterations N.
- Batch GP-Q: From the training set T obtained in the training phase, the input to the GP extracted as dX, is the set of all state-action pairs (s, a). The target value is the sum of current reward r and the maximum value predicted by the GP for the next state s' over all actions $a \in A$ (Returned by the function findMax(GP,s') line 20). The GP is updated with the input dX and the target vector tX (refer to line 18) and the loop is continued till the termination condition is not reached.

3.4 Simulations and Results

The performance of a batch-GPQ algorithm is compared with a naive Q-learning algorithm on the simulators shown in Figure **??**. Similar to the Figure **??**, we use 3 lasers on the robot to gather the distance data. The goal of the robot is to navigate in the environment without

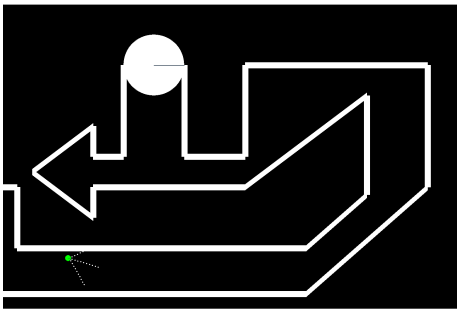
```
1  trainingPhase()
       Input      : S is a set of states
                    A is a set of action
       Local data: store the training data T := {s, a, r, s'},
                    previous state s
                    previous action a
                    ,current reward r
                    next state s'
       Initialize : T ← empty
2      for i ∈ (1, N) do
3          s ← current state
4          a ← random action
5          Apply a for a Δt time
6          s' ← observed state
7          r ← observed reward
8          T ← T ∪ {(s, a, r, s')}
9      end
10 batchGPQ()
       Input      : T := (s, a, r, s') is the training set recorded during the training phase
       Initialize : Gaussian Process GP with RBF kernel of appropriate length scale
11     while Termination condition not reached do
12         dX ← input to the GP
13         tX ← output of the GP.
14         for i ∈ (1, N) do
15             dX ← dX ∪ (s_i, a_i)
16             tX ← r_i + findMax(GP, s')
17         end
18         updateGP (dX, tX)
19     end
20 findMax(GP, s')
       Initialize : max ← 0
21     for a ∈ A do
22         temp ← gpPredict (s',a)
23         if temp > max then
24             max ← temp
25     end
26     return max
```

Algorithm 2: Batch GPQ Algorithm

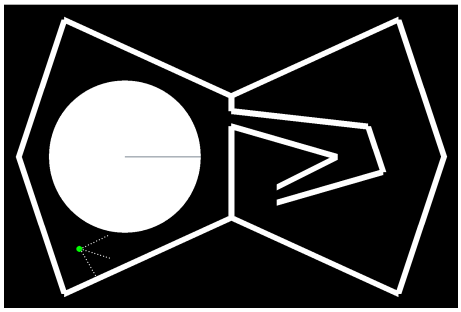


(a) Environment 1

(b) Environment 2



(c) Environment 3



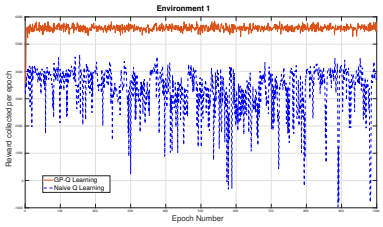
(d) Environment 4

Figure 3.4: Simulator Environments

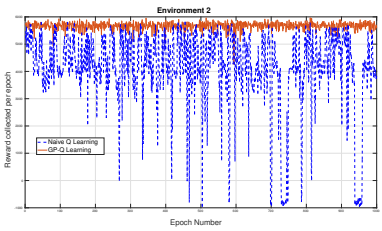
colliding into the obstacles. A reward of -5 is given on collision and a positive reward which is a function of the sum of the laser values is awarded on being able to avoid colliding into obstacles.

3.4.1 Reward as a function of time

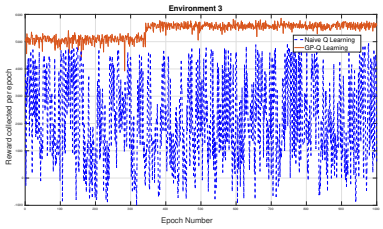
In order to compare the performance of the algorithms, we use *reward* as a metric of comparison. The plot shown in Figure **??** , shows the reward obtained over time for the naive Q learning and the end to end GP-Q learning algorithm. After every epoch, the reward obtained by the agent is calculated.



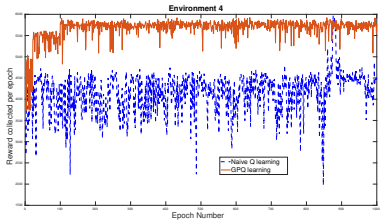
(a) Reward collected in Environment 1



(b) Reward collected in Environment 2



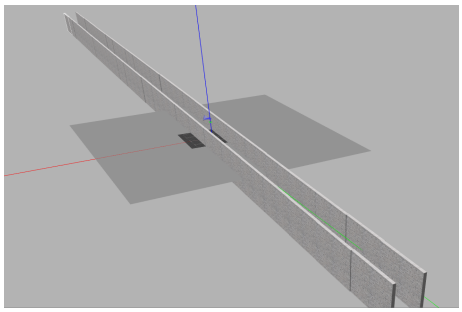
(c) Reward collected in Environment 3



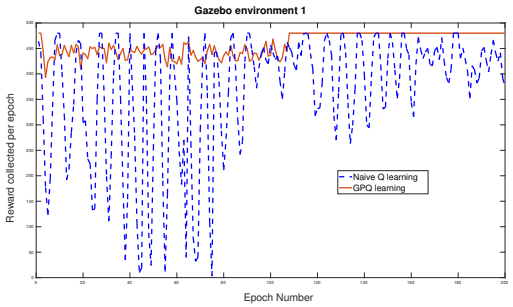
(d) Reward collected in Environment 4

Figure 3.5: Reward obtained over time for environments shown in Figure **??**. One *epoch* is a set of 200 actions.

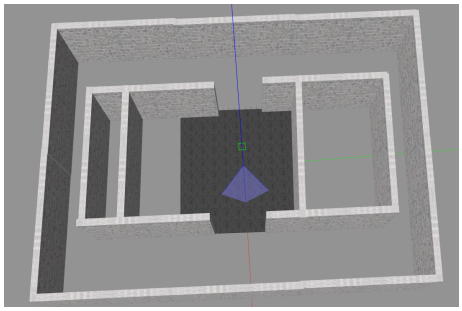
The GP learned in the basic kinematic simulators shown in Figure **??** is used directly to find the optimal policy to navigate the Gazebo Robot Simulator environments and the performance of the GP algorithm is compared with a naive Q learning algorithm by the reward plots shown in Figure **??**. It is evident from the plots that the GPQ algorithm performs better than the naive Q learning algorithm.



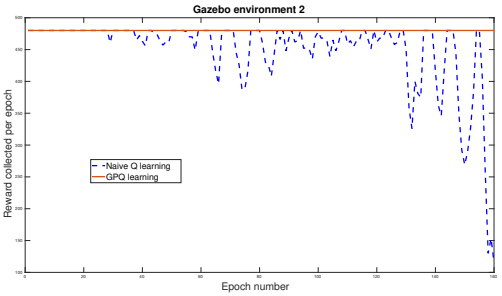
(a) Gazebo environment 1



(b) Reward collected in Gazebo environment 1



(c) Gazebo environment 2



(d) Reward collected in Gazebo environment 2

Figure 3.6: Gazebo Robot Simulator environments and the reward collected over time. One *epoch* is a set of 20 actions.

Chapter 4

The GP-MFRL Algorithm

In this chapter,the use of GP regression to learn the transition function is described and the details about the GP-MFRL algorithm – the main work of this thesis are provided. The simulator system setup is discussed in the section

4.1 Multi-Fidelity Reinforcement Learning

We build our work upon a recently proposed Multi-Fidelity Reinforcement Learning (MFRL) algorithm by Cutler et al. [**?**]. MFRL leverages multiple simulators to minimize the number of real world (*i.e.*, highest fidelity simulator) samples. The simulators denoted by $\Sigma_0, \ldots, \Sigma_D$, have increasing levels of fidelity with respect to the real environment. For example, Σ_0 can be a simple simulator that models only the robot kinematics, Σ_1 can model the dynamics as well as kinematics, Σ_2 can additionally model the wind disturbances, and the highest fidelity simulator can be the real world (Figure **??**).

MFRL differs from transfer learning [**?**] where transfer of parameters is allowed only in one direction. The MFRL algorithm starts in Σ_0. Once it learns an optimal policy in Σ_0, it switches to a higher fidelity simulator. If it observes that the policy learned in lower fidelity simulator is no longer optimal in the higher fidelity simulator, it either switches back to a lower fidelity simulator or stays at the same level. It was shown that the resulting algorithm has polynomial sample complexity and minimizes the number of samples required from the highest fidelity simulator.

The original MFRL algorithm uses Knows-What-It-Knows (KWIK) framework [**?**] to learn the transition and reward functions in each level. The algorithm essentially maintains a mapping from a state-action pair to the learned reward and the next state. The reward and the transition for each state-action pair is learned independently of others. While this is reasonable for general agents, in case of planning for robots we can exploit the spatial

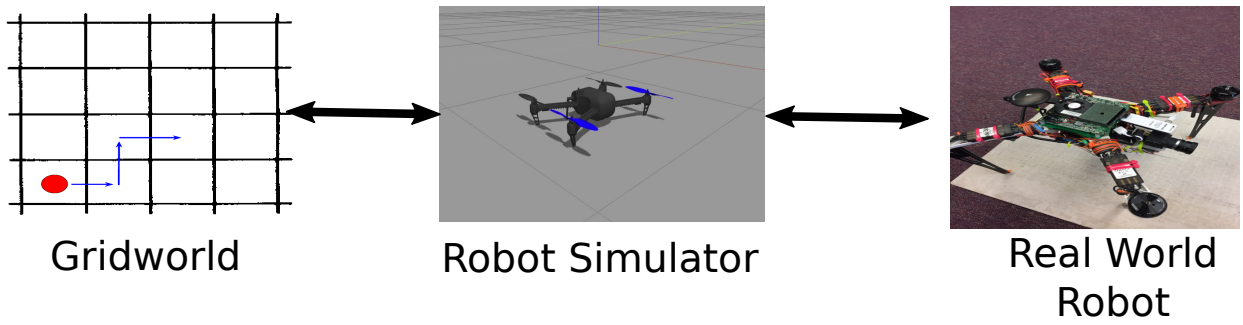


Figure 4.1: MFRL framework: First simulator captures only gridworld movements of a point robot while second simulator has more fidelity using a physics simulator. Control can switch back and forth between simulators and real environment which is essentially the third simulator in the multi-fidelity simulator chain.

correlation between neighboring state-action pairs to speed up the learning. Our main contribution in this paper is to show how to use Gaussian Process (GP) regression to learn the transition function in an MFRL framework using fewer samples.

GPs are commonly used to learn transition models for agents moving in the real world [**?**] and have been used in RL to learn the transition function [**?**], the reward function [**?**] and the value function [**?**]. GPs can predict the learned function for any query state-action pair, and not just for the discretized set of state-action pairs used when planning. In MFRL, the state space of Σ_i is a subset of the state space of Σ_j for all $j > i$. Therefore, when the MFRL algorithm switches from Σ_i to Σ_{i+1} it already has an estimate for the transition function for states in $\Sigma_{i+1} \setminus \Sigma_i$. Thus, GPs are particularly suited for MFRL which we verify through our simulation results.

4.2 Learning Transition Dynamics as a GP

A Markov Decision Processes (MDP) [**?**] is defined by a tuple: $\langle S, A, \mathcal{R}^a_{ss'}, \mathcal{P}^a_{ss'}, \gamma \rangle$. S and A are the set of states and actions respectively. $\mathcal{R}^a_{ss'}$, referred to as reward dynamics, defines the reward received by the agent in making a transition from state s to s' while taking action a and $\mathcal{P}^a_{ss'}$ is the probability of making this transition (also referred to as transition dynamics).

Generally the agent does not know the reward it will receive after making a transition nor does it know the next state it will land in. The agent learns these parameters through interactions with the environment which is subsequently used to plan an optimal policy to earn the maximum expected reward, *i.e.*, $\pi^* : S \to A$.

RL algorithms are broadly classified into *model-free* learning and *model-based* learning as seen in section **??**. Approaches that explicitly learn transition dynamics and/or reward dynamics

of an environment are known as model-based learning [?, ?]. The learned transition and reward dynamics can then be used to find the optimal policy using, for example, policy iteration or value iteration [?] which are often referred to as *planners*. In contrast, Strehl et al. [?] presented a model-free algorithm wherein the agent directly learns the value function and obtains the optimal policy. In this paper, we focus on model-based approaches and use GP regression to learn the transition dynamics.

Rasmussen and Kuss [?] showed how to use GPs for carrying out model-based RL. They assumed that the reward dynamics are known and the transition and value function was modeled as a GP. We use the same assumption for ease of exposition. However, assuming reward dynamics to be known is not a critical requirement. In fact, reward dynamics can also be easily be modeled as GPs.

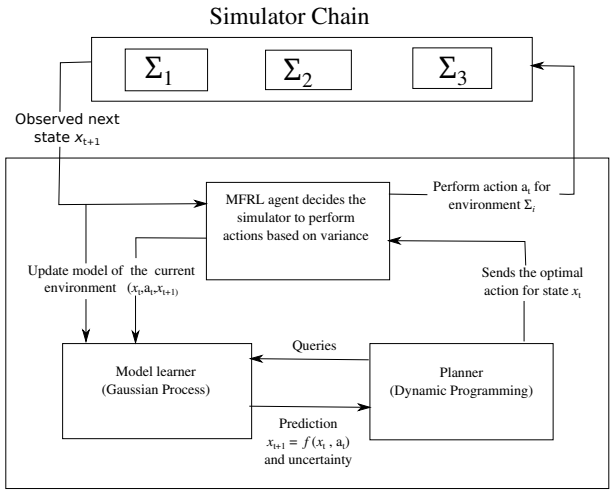


Figure 4.2: Overview of the GP-MFRL algorithm

We observe a number of transitions: $\mathcal{D} = \{(\mathbf{x}_t, a_t, \mathbf{x}_{t+1})\}$. Let $\mathbf{x}_{t+1} = f(\mathbf{x}_t, a_t)$ be the (unknown) transition function that must be learned. Our goal is to learn an estimate $\tilde{f}(\mathbf{x}, a)$ of $f(\mathbf{x}, a)$ in as few samples in $\mathcal{D}$ as possible. We can then use this estimated $\tilde{f}$ for unvisited state-action pairs (in place of f) during value iteration to learn the optimal policy. f can also be a stochastic transition function, in which case, the GP estimate gives the mean and the variance of this noisy transition function. For a given state-action pair (s, a), the estimated transition function is defined by a normal distribution with mean and variance given by:

$$\mu_{(s,a)|\mathcal{D}} = \mathcal{K}_{(s,a)\mathcal{D}}\mathcal{K}_{\mathcal{D}\mathcal{D}}^{-1}\vec{\mathcal{X}}_{\mathcal{D}} \tag{4.1}$$

$$\sigma^2_{(s,a)|\mathcal{D}} = \mathcal{K}\{(s,a),(s,a)\} - \mathcal{K}_{(s,a)\mathcal{D}}\mathcal{K}_{\mathcal{D}\mathcal{D}}^{-1}\mathcal{K}_{(s,a)\mathcal{D}} \tag{4.2}$$

where $\mathcal{K}$ is the kernel function.

GP regression requires a kernel which encodes the correlation between the values of f at two points in the state-action space. Choosing a right kernel is the most crucial step in

implementing GP regression. We choose the Radial Basis Function (RBF) kernel for our implementation since it models the spatial correlation we expect to see in an aerial robot system well. However, any appropriate kernel can be used in our algorithm depending on the environment to be modeled.

RBF has infinite dimensional feature space and satisfies the Lipschitz smoothness assumption. It can be defined as follows: for two points $\mathbf{x}$ and $\mathbf{x'}$,

$$k(\mathbf{x}, \mathbf{x'}) = \exp\left(- \frac{||\mathbf{x} - \mathbf{x'}||^2}{2\sigma^2} \right) \tag{4.3}$$

where $||\mathbf{x} - \mathbf{x'}||^2$ is the squared Euclidean distance and σ is a hyperparameter for the kernel often known as *characteristic length-scale*. Here, $\mathbf{x}$ represents a point in the joint state-action space.

Instead of using GPs to predict the next state, we use it to predict the velocity with which the robot will move when a given action a_t is applied at a state s_t. Learning the velocity vector helps in transitioning between simulators as the size of the state space itself may be different. For example, one can construct a multi-fidelity simulator where Σ_0 is a $n \times n$ grid, Σ_1 is a denser $2n \times 2n$ grid, and so on. An action in Σ_0 moves the robot one unit whereas the same action in Σ_1 moves the robot only 0.5 units. By learning the velocity instead of the next state, we can scale the learned velocity function to easily compute the transition function in any Σ_i as:

$$\vec{\mathcal{V}}\ (s_t, a_t) = \frac{s_{t+1} - s_t}{\Delta_i} \tag{4.4}$$

where Δ_i is the time scaling of a simulator. If the state spaces of all simulators are the same, then one can use GPs to predict the next state instead of the velocity vector.

We train two GP regressions, $f_x, f_y : \mathbb{R}^4 \to \mathbb{R}$, assuming independence between the two output dimensions. Let (x_i, y_i) be the current state of the agent. Actions actions are represented using a tuple (a_x, a_y) where a_x and a_y can take the values between 0 or 1.

The GP prediction is used to determine the transitions, $(x_i, y_i, a_x) \to x_{i+1}$ and $(x_i, y_i, a_y) \to y_{i+1}$ where (x_{i+1}, y_{i+1}) is the predicted next state with variance σ_x and σ_y respectively. Value of hyperparameters is estimated by gradient descent by optimizing the maximum likelihood estimate of a training data set.

4.3 GP-MFRL Algorithm

Using multiple approximations of real world environments has previously been considered in the literature [**?**, **?**]. Cutler et al. used model-based R-Max algorithm to reduce the number of samples using MFRL framework [**?**]. We use GP regression to further bring down the empirical sample complexity of MFRL framework.

Algorithm **??** gives the details of the proposed framework. As illustrated in Figure **??**, there are two main components of GP-MFRL: (1) Model Learner; and (2) Planner. The model learner in our case is the GP-regression described in the previous subsection. We use value iteration [**?**] as our planner to calculate the optimal policy on learned dynamics of environment.

An *epoch* measures the time span between two consecutive switches in the simulators. Before executing an action, the agent checks (Step 4) if it has a sufficiently accurate estimate of the transition dynamics for the current state-action pair in the lower fidelity simulator, Σ_{d-1}. If not, it switches to Σ_{d-1} and executes the action in the potentially less expensive environment. The function ρ^{-1} checks if the current state is also a valid state in the lower fidelity simulator.

We also keep track of the variance of the $\mathcal{L}$ most recently visited state-action pairs in the current epoch. If the running sum of the variances is below a threshold (Step 8), this suggest that the robot has found a good policy in the current simulator and it must advance to the next higher fidelity simulator.

Steps 12–16 in describe the main body where the agent computes the optimal action, executes it, and records the observed transition in $\mathcal{D}$. The GP model is updated after every n_U iterations (Step 17). In the update, we recompute the hyper-parameters until they converge.

A new policy is computed every time the robot reaches the goal state (Step 21). If the robot is in the highest fidelity simulator, we also check if the policy has converged by checking if the maximum change in the value function is less than a threshold (Step 22). If so, we terminate the learner.

4.4 Simulation Results

We demonstrate the GP-MFRL algorithm in a simulator chain consisting of a virtual grid-world environment and the Gazebo robot simulator [**?**]. The setup is shown in Figure **??**. The simple grid-world agent operates in a 21×21 grid. The agent receives a reward of $+50$ at the goal location, and -1 for all other states. If the agent hits the obstacles, it gets the reward of -20. In each time step, an agent can move in one of the four directions viz. up, down, left and right. We add a Gaussian noise of σ to the actual transition to represent stochastic environments. The Gazebo simulation setup consists of a quadrotor with PX4 autopilot running in software–in–the–loop (SITL) mode. The PX4 SITL interfaces with the Robot Operating System [**?**] via the `mavros` node.

The code is written in python and uses scikit-learn [**?**] to implement GP-regression. The code is available online at `https://github.com/raaslab/gp_gazebo`.

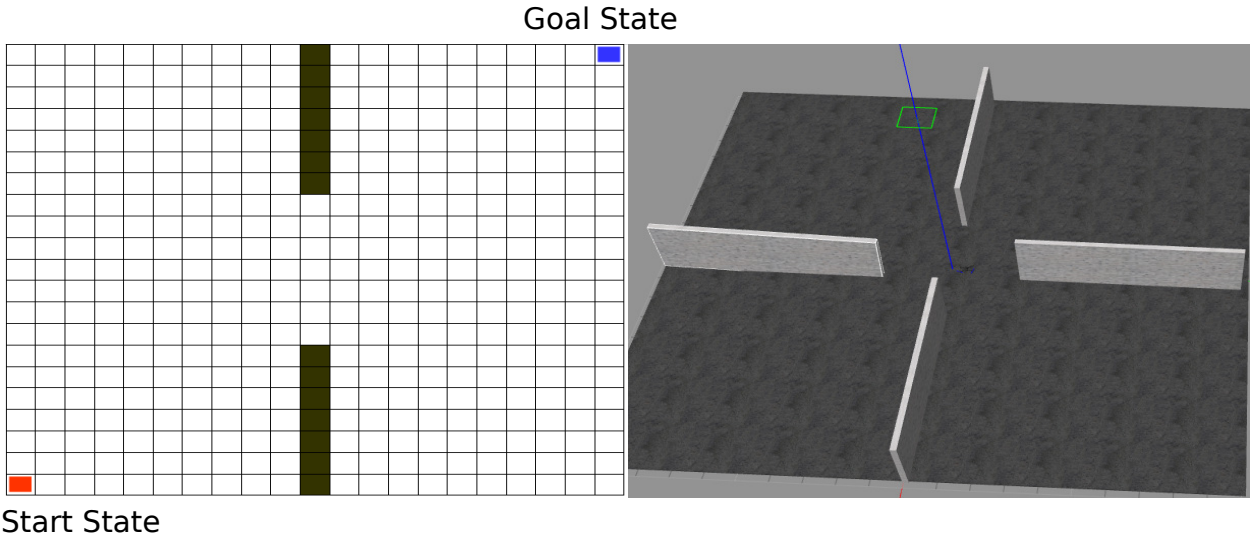


Figure 4.3: The environment setup for a multi-fidelity simulator chain. The simple gridworld environment has two wall obstacles whereas the gazebo environment has four wall obstacles as shown.

Figure **??** shows the switching between the simulators for one run of the GP-MFRL algorithm on the environment shown in Figure **??**. It can be seen that the agent switches back and forth between the two simulators unlike unidirectional transfer learning algorithms. In the rest of the simulations we study the effect of the parameters used in GP-MFRL and the fidelity of the simulators on the number of samples till convergence.

Input : A simulator chain,
Confidence parameter ψ for (s, a),
History Length $\mathcal{L}$,
Confidence Ψ,
State mapping ρ,
Reward dynamics $\mathcal{R}^a_{ss'}$
Update rate n_U
Initialize: Transition dynamics $\mathcal{P}^a_{ss'}$;
$d = 1$;
$\mathcal{V}^*_d(s) \leftarrow$ `Planner` $(\mathcal{P}^a_{ss'})$

```
Learner()
```
while true **do**
  $a^*_t \leftarrow \text{argmax}_a\ \mathcal{V}^*_d(s_t)$;
  if $\sigma(\rho^{-1}(s_t, a^*_t)) \geq \psi\ \bigwedge\ d > 1$ **then** `// Return to level` $d-1$
    $d \leftarrow d - 1$;
    `epochLength` $\leftarrow\ 0$
  end
  if $\sum_{i=t-\mathcal{L}}^{t-1} \sigma(s_i, a^*_i) \leq \Psi\ \bigwedge$ `epochLength` $\geq \mathcal{L}$ **then**
    $d \leftarrow d + 1$ (Move up the simulator) ;
    `epochLength` $\leftarrow\ 0$;
  end
  $a^*_t \leftarrow \text{argmax}_a\ \mathcal{V}^*_d(s_t)$;
  Execute a^*_t and store observed s_{t+1}, r_{t+1} ;
  `epochLength` ++ ;
  $\mathcal{D}_t = \mathcal{D}_t \cup (s_t,\ a^*_t,\ s_{t+1})$;
  $s_t \leftarrow s_{t+1}$;
  if `epochLength` is multiple of n_U **then**
    $\mathcal{P}^a_{ss'} \leftarrow$ `UpdateGP`$(\mathcal{D}_t)$;
  end
  if s_t is Goal state **then**
    $\mathcal{V}_f(s) \leftarrow$ `Planner`$(\mathcal{P}^a_{ss'})$;
    if $\max_s\ \mathcal{V}_f(s) - \mathcal{V}_0(s) \leq 10\ \% \bigwedge d == D$ **then**
      break the loop ;
    end
    $\mathcal{V}_0(s) \leftarrow \mathcal{V}_f$;
  end
  $t \leftarrow t + 1$;
end

```
Planner()
```
Initialize: $\mathcal{V}(s) = 0,\ \forall(s, a)$
$\Delta = \infty$
while $\Delta > 0.1$ **do**
  for *every s:* **do**
    $temp \leftarrow \mathcal{V}(s)$;
    $\mathcal{V}(s) \leftarrow \max_a \sum_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma\mathcal{V}(s')]$;
    $\Delta \leftarrow \max(0,\ |temp - \mathcal{V}(s)Init|)$
  end
end
return $\mathcal{Q}(s, a)$

Algorithm 3: GP-MFRL Algorithm

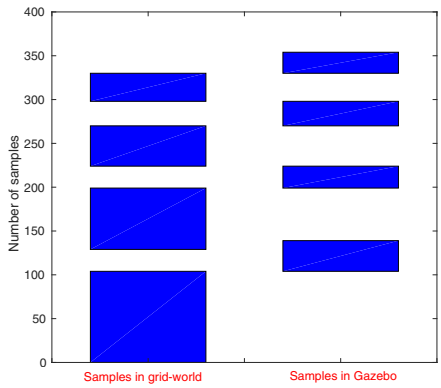
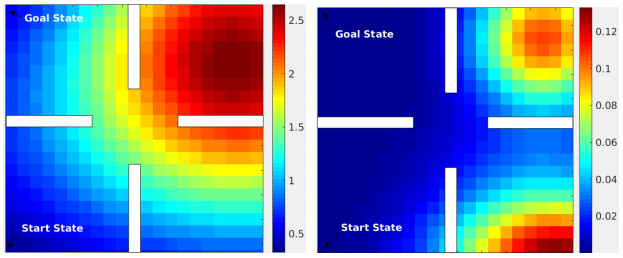


Figure 4.4: The figure represents the samples collected in each level of simulator for a 21×21 grid in a simple grid-world and Gazebo environments. Ψ and ψ were kept 0.4 and 0.1



(a) After Initial Training (b) After Convergence

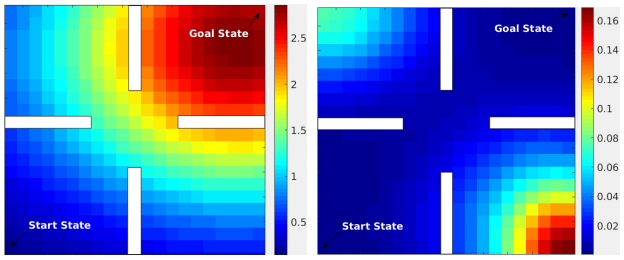
Figure 4.5: Variance plot for 21×21 multi-fidelity environment after transition dynamics initialization and after algorithm has converged

4.4.1 Representative simulations

We first present three representative scenarios to observe the qualitative performance of the GP-MFRL algorithm. Specifically, we consider three instances and show how the variance evolves over time as more samples are collected. Recall that the main advantage with using GPs is that it allows for quick generalization of observed samples to unobserved state-action pairs.

To demonstrate how variance of the predicted transition dynamics varies from the beginning of experiment to convergence, we plot "heatmaps" of the variance. The GP prediction for a state-action pair also gives the variance, σ_x and σ_y, respectively for the predicted state. The heatmap shows $\sqrt{\sigma_x^2 + \sigma_y^2}$ for the optimal action at every state as returned by the Planner.

Figures **??** and **??** show the heatmaps at the start and convergence for the same environment but with different start and goal positions. As expected, the variance along the optimal (*i.e.*, likely) path is low whereas the variance for states unlike to be on the optimal path from start to goal remains high.



(a) After Initial Training (b) After Convergence

Figure 4.6: Variance plot for 21 × 21 multi-fidelity environment after transition dynamics initialization and after algorithm has converged

A more interesting case is presented in Figure **??**. Even though there's a path available to reach the goal from the right of wall A, the agent explores that region less than the region near the walls B, C and D (indicated in dark blue showing less variance). This is due to the fact that, the transition dynamics learned in the lower fidelity simulator is used in the higher fidelity simulator leading to lesser exploration of the regions which are not along the optimal path.

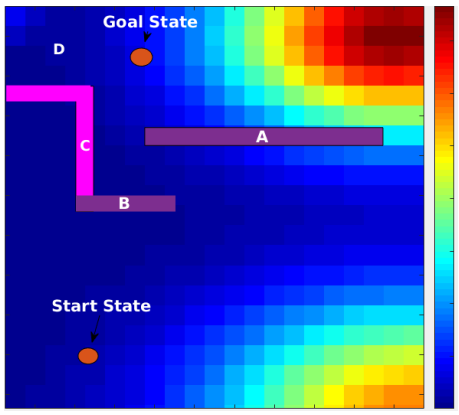


Figure 4.7: Variance plot for 21 × 21 multi-fidelity environment after the algorithm has converged. Walls A and B are only present in the grid-world simulator, whereas all four walls are present in the Gazebo simulator.

4.4.2 Effect of fidelity on the number of samples.

We first study the effect of varying fidelity on the total number of samples and the fraction of the samples collected in the higher fidelity simulator. Our hypothesis is that having learned the transition dynamics in the gridworld, the agent will need fewer samples in the higher fidelity Gazebo simulator to find the optimal policy. However, as the fidelity of the first

simulator decreases, we would need more samples in Gazebo.

In order to validate this hypothesis, we varied the noise parameter used to simulate the transitions in the gridworld. The transition model in Gazebo remains the same. The total number of samples collected increases as we increase the noise in gridworld (Figure **??**). As we increase the noise in the first simulator, the agent learns less accurate transition dynamics leading to collection of more number of samples in the higher fidelity simulator. Not only does the agent need more samples, the ratio of the samples collected in the higher fidelity simulator to the total number of samples also increases (Figure **??**).

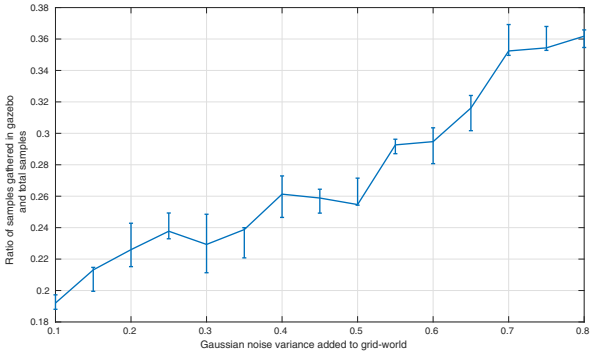


Figure 4.8: As we make first simulator more inaccurate by adding noise, the agent tends to gather more samples in second simulator

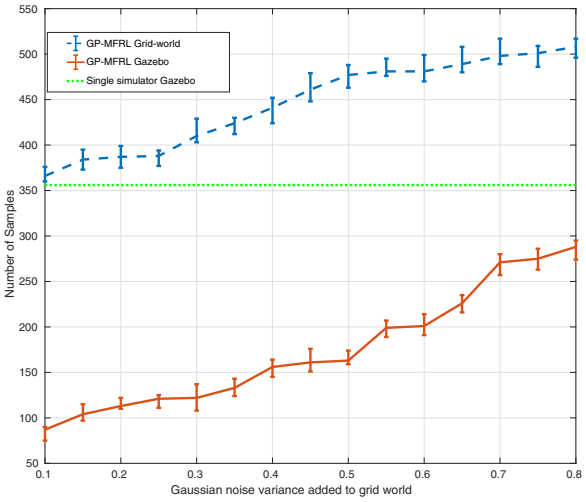


Figure 4.9: Ratio of samples gathered in the second simulator to the total samples gathered increases with inaccuracy in the first simulator. The reference line depicts the average number of samples gathered over 10 runs when only Gazebo simulator was present.

4.4.3 Effect of the confidence parameters.

The GP-MFRL algorithm uses two confidence parameters, ψ and Ψ, which are compared against the variance in the transition dynamics to switch to a lower and higher simulator, respectively. Figure **??** shows the effect of varying the two parameters on the ratio of number of samples gathered in the Gazebo simulator to the total number of samples. As expected, increasing ψ or decreasing Ψ leads to more samples being collected in the higher fidelity simulator.

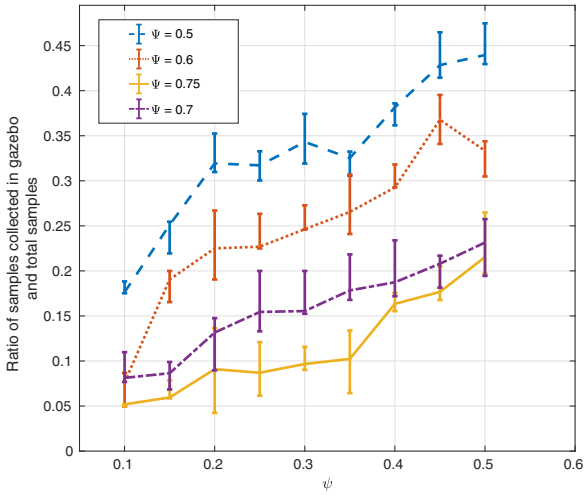


Figure 4.10: Ratio of samples gathered in second simulator vs. total samples gathered as we change the threshold or confidence parameters of the two simulators.

4.4.4 Comparison with R-max MFRL

Figure **??** shows the comparison between performance of GP-MFRL algorithm with the existing MFRL algorithm [**?**], GP-MFRL algorithm only in the highest fidelity simulator and Rmax algorithm running only in the highest fidelity simulator. The experiments are performed in the environment same as the one used in Figure **??**. As expected, the GP-MFRL algorithm performs better than the existing MFRL algorithm, [**?**].

4.5 Conclusion

The GP-MFRL algorithm provides a general RL technique that is particularly suited for robotics. An extension to the existing work would be implementing the GP-MFRL algorithm on an actual quadrotor as the highest-fidelity simulator to demonstrate the utility of GP-

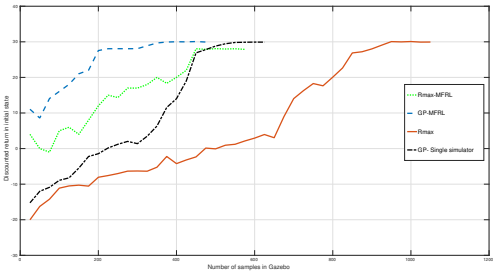


Figure 4.11: Discounted return in the start state Vs. the number of samples collected in the highest fidelity simulator.

MFRL. In this thesis, it is shown empirically that, GP-MFRL finds optimal policies using fewer samples than MFRL algorithm.

Chapter 5

Bridge Inspection

5.1 Background

When it comes to inspecting bridges, UAVs are often considered as harbingers, a truly disruptive technology capable of giving engineers eyes in the hardest to reach places, without the need for expensive access vehicles or potentially dangerous rigging or ladders or harnesses. A recent paper [**?**] discusses the application of UAVs for visual inspection and damage detection of civil structures. The quality of such pictures and videos taken by the UAV strongly depends on various factors viz. lighting conditions, distance from the structure, vehicle motion influenced by environmental effects etc. It is very important for the UAVs to be equipped with sophisticated control algorithms and sensors to be able to provide reliable data which can be conclusively used for inspection and analysis of damage.

5.1.1 Conventional Inspection of Bridges

Figure **??** shows typical inspection units like under-bridge units or truck cranes. Inspection units are in most cases expensive custom products. Often, specially trained staff, like industrial climbers, can get access to special parts of the structure but they can rarely evaluate what influences detected damages have on these structures. Therefore, they can only take Photos or Videos of the concerned part of the structure, which must be analyzed by civil engineers off-line.

It is often dangerous to use such large truck cranes. Figure **??** shows an unfortunate incidence where the inspection truck has tipped over causing a life threating situation for the people involved.

(a) Truck Crane (b) Under Bridge Inspection unit

Figure 5.1: Conventional inspection units for Bridge Inspection: (left,Bridge Inspection platform with a truck crane) , A-30 Hi-Rail Under Bridge Units(right,N.E. Bridge Contractors Inc.)

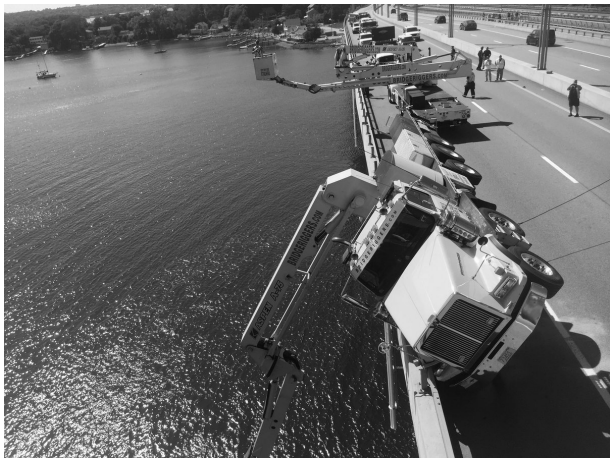


Figure 5.2: Bridgeriggers' Aspen A-75 under-bridge inspection crane overturns on Sakonnet River Bridge in Rhode Island; August 30, 2016

5.1.2 UAVs in Bridge Inspection

UAVs often have following advantages in comparison to the conventional bridge-inspection methods.

- UAVs only need an operator on the ground for controlling the flight and the camera. A more advanced scenario includes *autonomous* UAVs navigating around the structures while collecting the required data such as pictures, videos etc. reference to a later section where we talk about challenges of navigating around such structures.
- UAVs can be used in high-risk situations without endangering human lives. Situations like **??** could be avoided to a certain extent.
- UAVs are capable of fast real time data acquisition and the storage of all the relevant flight data.
- Overall, they often turn out to be lower in costs compared to the large custom inspection units.

As they say *Every good thing comes with a price*, UAVs are no different. Despite the obvious advantages, UAVs have some essential limitations.

- Due to the small payload capacity of the UAVs, only small format and light digital compact cameras can be used for visual inspection.
- The smaller payload also affects the battery size leading to shorter flight times.
- Due to unexpected flight situations or due to unreliable GPS-signal strength, a change from the autonomous flight mode into a manual mode is required which demands a skillful pilot to override the control.
- Currently, UAVs are not equipped with effective collision avoidance systems.

5.2 Present Work

We performed a few experiments in order to analyze the limitations and challenges of performing autonomous navigation of UAVs around bridges. We performed several manual flights of a UAV on the Smart Road Bridge (the tallest state-maintained bridge in Virginia). The goal of the experiment was to capture visuals of the inside as well as the outside of the bridge for analysis of the structure and to analyze challenges in order to improve the system for fully autonomous flights.

5.2.1 The System Setup

DJI model F450 [**?**] was used as shown in the figure **??**. The UAV is equipped with sensors, controller and an on-board computer.

Figure 5.3: UAV used for the experiments

The specific components are listed below.

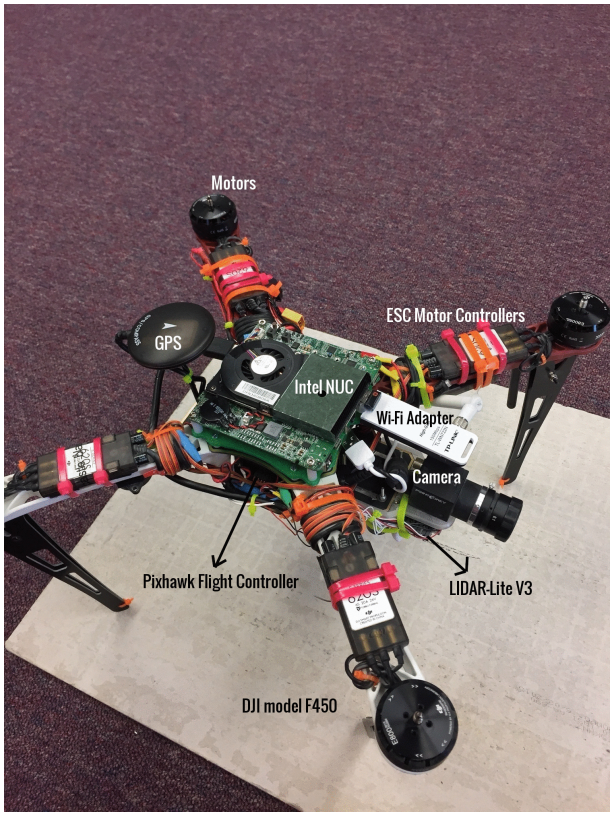


Figure 5.4: System Components

- **Camera**: The camera used to capture the visuals is [**?**] as shown in figure **??**. The resolution of the images is 1600×1200 and can capture images upto 59 FPS.

Figure 5.5: Flea3 2.0 MP Color USB3 Vision (e2v EV76C5706F)

- **Intel NUC**: Intel NUC5I7RYH Mini PC NUC Kit as shown in figure **??**.

Figure 5.6: Intel NUC NUC5I7RYH

Specifications are as follows:

  - 5th Generation Intel Core *i*7-5557U processor
  - Intel Iris Graphics 6100
  - Mini HDMI & Mini Display Port
  - Internal support for M.2 SSD card & SATA3 for 2.5“ HDD/SSD

- **Pixhawk-Flight Controller**: Pixhawk is an independent, open-hardware project which mainly aims at providing high-end autopilot hardware to the academic, hobby and industrial communities at low costs as shown in figure **??**.The Pixhawk autopilot module runs a very efficient real-time operating system (RTOS), which provides a POSIX-style environment.

Figure 5.7: Pixhawk Flight Controller

Some of the specifications are as follows:

- 68 MHz Cortex M4F CPU (256 KB RAM, 2 MB Flash)
- Sensors: 3D Accelerometer / Gyroscope / Magnetometer / Barometer
- Integrated backup, override and failsafe processor with mixing
- MicroSD slot, 5 UARTs, CAN, I2C, SPI, ADC, etc

- **LIDAR**: The LIDAR-Lite v3 is a compact, high-performance optical distance measurement sensor from Garmin™ as shown in figure **??**.

Figure 5.8: LIDAR-Lite v3

The specifications are as follows:

  - Range: 0-40m Laser Emitter
  - Accuracy: +/- 2.5cm at distances greater than 1m
  - Power: 4.755V DC; 6V Max
  - Current Consumption: 105mA idle; 130mA continuous
  - Rep Rate: 1500Hz
  - Laser Wave Length/Peak Power: 905nm/1.3 watts
  - Beam Divergence: 4m Radian × 2m Radian
  - Optical Aperture: 12.5mm
  - Interface: I2C or PWM

- **GPS**: GPS used is Ublox Neo-M8N GPS with Compass as shown in figure **??**.

Figure 5.9: Ublox Neo-M8N GPS

The specifications are as follows:

  - 167 dBm navigation sensitivity
  - Navigation update rate up to $10H_z$
  - Cold starts: 26s
  - $25 \times 25 \times 4$ mm ceramic patch antenna
  - Rechargeable 3V lithium backup battery
  - Low noise 3.3V regulator
  - Diameter 60mm total size, 32 grams with case.

- **Motors and ESC Motor Controllers**: E800 Tuned Propulsion System is designed for multi-rotor copters. Shown in figure **??**

Figure 5.10: E800 Tuned Propulsion System by DJI

5.2.2 Camera Software and ROS nodes

The ROS-flea3 camera driver enables publishing of images on ROS topics which can in turn be viewed on rqt image view tool. The images can be viewed real time and are also stored in a rosbag. The screen capture of an instance of the camera view and ROS node running the rosbag to record to images is shown in figure

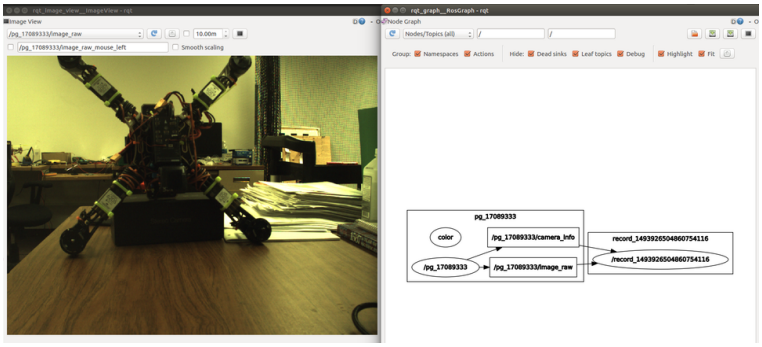


Figure 5.11: Camera view and ROS-bag record

5.2.3 Flights without GPS signal

While the UAV is flying indoors or inside the bridge, it does not get a GPS signal for a better position estimate. Manual control is very difficult due to the smaller space and narrow pathways. In chapter **??**, we discussed the advantages of effectively learning obstacle avoidance or wall-following as a skill using the on-board sensor data. It is very useful in these scenarios where the UAV needs to rely on it's on-board sensors to perform a particular task simultaneously trying to avoid obstacles or follow a wall. Figure **??** indicates one such scenario where the UAV needs to take pictures inside the structures.

Figure 5.12: Indoor flight of the UAV visually inspecting the structure

5.2.4 Outdoor Inspection

Similar to the indoor bridge inspection, while flying close to the bridge surfaces, the UAV does not get reliable GPS signal or any other localization inputs. It is vital for the robot to have learned the skill of flying close to the surfaces and following certain trajectories using on board laser sensor data. As iterated multiple times in this thesis, while learning in the real world, collecting negative samples (Colliding with the wall/surface in this scenario) can lead to catastrophic results. It is important for the robot to learn a skill in a simulator environment before executing the required tasks in the real world. Figure **??** shows a picture of one of such flights for the outdoor visual inspection of the bridge.

Figure 5.13: Outdoor flight of the UAV visually inspecting the structure

Chapter 6

Conclusion and Future Research

The GP-MFRL algorithm provides a general RL technique that is particularly suited for robotics. Our immediate work focuses on extending the End-to-End GPQ algorithm to a multi-fidelity simulator framework. Implementing the algorithm on an actual quadrotor as the highest fidelity simulator to demonstrate the utility of GP-MFRL is the immediate goal. We analyze empirically that GP-MFRL and GPQ find optimal policies in fewer samples than their naive counterparts. One of the main tasks is to find the theoretical bounds on the sample complexity of GP-MFRL.

In End-to-End GPQ algorithm, as the number of observations increase, the time taken to perform GP updates also increases with a cubic order. We can use adaptive sample selection techniques [**?**] as well as numerical optimization techniques [**?**] to speed up this process. Our ongoing work on sparse online Gaussian Processes [**?**] should be able to overcome the limitations for larger data sets in order to perform online GP prediction of Q values instead of off-line batch predictions.

Theoretical and simulation results are useful in defining the performance limits and this level of abstraction is useful in providing insight to the research direction and making progress. However, it is important to bridge the gap between these results and practical limitations in order to deploy intelligent robots in realistic environments. The motivating example of bridge inspection is one such example of a real environment. Our goal is to be able to successfully perform autonomous navigation of UAVs in such environments.