November 2017

ADAFT: A RESOURCE-EFFICIENT FRAMEWORK FOR ADAPTIVE FAULT-TOLERANCE IN CYBER-PHYSICAL SYSTEMS

Ye Xu

ADAFT: A RESOURCE-EFFICIENT FRAMEWORK FOR ADAPTIVE FAULT-TOLERANCE IN CYBER-PHYSICAL SYSTEMS

A Dissertation Presented

by

YE XU

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2017

Electrical and Computer Engineering

ADAFT: A RESOURCE-EFFICIENT FRAMEWORK FOR ADAPTIVE FAULT-TOLERANCE IN CYBER-PHYSICAL SYSTEMS

A Dissertation Presented

by

YE XU

Approved as to style and content by:

Israel Koren, Co-chair

C. Mani Krishna, Co-chair

David Irwin, Member

Deepak Ganesan, Member

Christopher V. Hollot, Department Chair
Electrical and Computer Engineering

ABSTRACT

ADAFT: A RESOURCE-EFFICIENT FRAMEWORK FOR ADAPTIVE FAULT-TOLERANCE IN CYBER-PHYSICAL SYSTEMS

SEPTEMBER 2017

YE XU

B.Sc., NANJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Israel Koren and Professor C. Mani Krishna

Cyber-physical systems frequently have to use massive redundancy to meet application requirements for high reliability. While such redundancy is required, it can be activated adaptively, based on the current state of the controlled plant. Most of the time the physical plant is in a state that allows for a lower level of fault-tolerance. Avoiding the continuous deployment of massive fault-tolerance will greatly reduce the workload of CPSs. In this dissertation, we demonstrate a software simulation framework (AdaFT) that can automatically generate the *sub-spaces* within which our adaptive fault-tolerance can be applied. We also show the *theoretical benefits* of AdaFT, and its *actual implementation* in several real-world CPSs.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

Dramatic changes have occurred over the past few years in cyber-physical systems (CPSs). Such systems range in complexity from simple small-scale to complex large-scale. The traditional approach to controlling CPSs has been to use a large number of controllers that interact with each other, and are dedicated to perform a certain subset of computational tasks. For example, an automotive application might have a micro-controller entirely dedicated to braking control, another dedicated to cruise control, while another subset may be dedicated to managing the entertainment system.

More recently, in an effort to provide increased reliability and reduce costs, designers have been turning to a more flexible approach, with a shared, integrated, computational platform. Such a platform is responsible for the totality of the control function; individual computers may be shared by different functions. The same physical computer can run widely varying tasks, whose importance may range from non-critical to life-critical. Tasks can be remapped from one processor to another depending on prevailing load conditions and the health of the processor. Such an approach, when handled correctly, yields a control structure that can degrade more gracefully and reliably when a core fails. As nodes fail, the still operational computational resources can focus on keeping the high-criticality tasks running, shedding the less vital functions as necessary. Recent years have seen the emergence of a rich theory of scheduling in such integrated, mixed-criticality platforms [9][57].

For obvious reasons, *fault-tolerance* (FT) is needed for life-critical applications. Traditional fault-tolerance that uses massive redundancy [23] can impose a considerable and unnecessary computational burden on the system. We need, therefore, to turn to adaptive fault-tolerance [24][25]. Here, the level of fault-tolerance provided is dynamically adapted, during operation, to the needs of the physical plant. These needs are calculated as a function of the current plant state and consequently, they vary by time and circumstance.

Adaptive fault-tolerance allows us to provide fault-tolerance on an as-needed basis and has the potential to reduce the size of the computational platform. It can also result in reduced processor operating temperatures. Since processor failure rates increase exponentially with operating temperature, this often has a significant impact on processor reliability.

In order to implement adaptive fault-tolerance, the controlled plant dynamics have to be analyzed along with domain-specific knowledge of safety requirements to indicate the appropriate level of fault-tolerance at any given time. In many cases, the controlled plant is operating in a deeply safe sub-space of the entire physical state space, and can survive one or more cyber-side failures even if the failures cause the actuator to produce a worst case incorrect control input to the plant. In such cases, there is no need to deploy the redundant copies of the control tasks. Therefore, there is a need to design a system that would take the physical side information into account, and determine when it is within this safe sub-space that would need less fault-tolerance.

1.2 Contributions

This dissertation describes a simulation framework, AdaFT, to accomplish the proposed adaptive fault-tolerance. This framework generates offline a table that allows the system, while in operation, to select the appropriate level of fault-tolerance. It does this by carrying out extensive offline analysis of the controlled plant dynamics and then uses standard artificial intelligence techniques to express them as simple selection rules. AdaFT also performs runtime workload tuning of the cyber side of the system, by introducing the concept

of *Quality of Control Constraints* (QoC Constraints) and using multiple versions of certain real-time control tasks, which can be switched during runtime to balance the QoC and the reliability of the system. Finally, the framework allows the designer to evaluate the impact on reliability of the thermal stress associated with the workload. Overall, AdaFT allows us to increase the system's lifetime, and conversely, for a given desired life time, it reduces the amount of redundancy required.

Our contributions are summarized as follows:

1. We introduce the concept of *Sub-spaces* of the physical plant, which partitions the entire state space into three parts that would need different levels of fault-tolerance deployment during runtime. We discuss in detail the algorithms to identify these sub-spaces, and how to handle various practical issues listed below.

2. The sub-spaces have to be expressed in a compact form so that, during operation, the system can rapidly determine which sub-space it is in, and adjust its redundancy level accordingly. In this dissertation we describe a classification process using machine learning techniques that generates classification rules which are lightweight enough to be used in practice.

3. We consider actuator noise, sensor noise and sensor failures. It is unrealistic to assume that the controlled plant has perfect information regarding its current state; this work explicitly allows for noise of any given distribution, and takes advantage of *Dynamic Bayesian Networks* (DBN) for inferencing. It also deals with temporary sensor failures and multiple sensors (sensor networks).

4. We introduce into our framework the capacity to handle reduced-order models (by using a fitted machine learning algorithm, rather than the full-level system dynamics model). This allows us to obtain the sub-spaces with greatly reduced overhead, even for the offline procedure.

5. In order to measure the reliability benefits of our approach, we use the concept of *Thermal Age Acceleration Factor* (TAAF), which can then be integrated over time to produce the long-term reliability benefits in terms of the *Mean Time To Failure* or MTTF. We provide a detailed discussion of TAAF and MTTF under different scenarios using our framework. Along the way, the long-term energy consumption of the traditional approach and our approach are also compared.

6. We design a *load tuner*, which is an essential component of our framework, that includes *number of copies determination* and *version selection*. Number of copies determination refers to selecting the appropriate fault-tolerance level according to the plant's current sub-space. Version selection means selecting the proper version of each control task, based on the QoC constraints.

7. We integrate into our framework several popular designs and modules of real-time control systems. One is the *real-time scheduling* for real-time systems. With this module, the user can see how different periods and possibly different response times of each control task can affect the size of each sub-space. Another one is the *Dynamic Voltage and Frequency Scaling* (DVFS) that can further improve the TAAF if the actual execution time of the control task is much less than its *worst case execution time* (WCET).

8. In addition to the TAAF and MTTF analysis, we show how our framework can also be used to determine the *hardware provisioning*.

9. Finally, we have put all these together into a framework called *AdaFT*, which can be used by other researchers. All the user has to do is to use the AdaFT API to specify the state equations and the control law of the controlled plant. Note that both linear and nonlinear plants can be handled.

This dissertation is organized as follows. In Chapter 2, the general background for real-time systems, fault-tolerance, and big data in embedded systems is presented. In Chap-

ter 3, the technical background of AdaFT and related work is discussed; this includes a state space approach to system control and a discussion of thermally induced circuit aging. These set the stage for the design of the software framework in Chapter 4. Chapter 5 includes several case studies as examples. Chapter 6 introduces the idea of using *reinforcement learning* to further improve the current AdaFT. Chapter 7 brings the dissertation to a close.

CHAPTER 2

BACKGROUND

2.1 Embedded System and Cyber Physical System

Cyber-Physical Systems (CPS) integrate multiple dynamics together including computing, networking, and physical processes. Embedded sensors, micro-controllers and networks sense, compute, and actuate the physical processes, with feedback loops where physical processes have an impact on computation and vice versa [18]. According to Lee, *et al.*, CPSs have an economic and societal potential which is vastly greater than what has been recognized [27]. Many countries have made major investments to develop the technology for various CPS solutions. CPS technology is built on top of the older discipline of embedded systems, where computers and software embedded in devices are not aimed at computation, but rather other specific purposes such as braking control in cars and navigation in aircraft. CPS researchers have developed modeling, design and analysis techniques for sophisticated integrated systems.

The term cyber-physical systems may be interpreted as the combination of the cyberspace and the physical processes [43]. The cyberspace part refers to techniques that have evolved over decades in computer science such as algorithms and data collection, transforming and analysis. The physical part of CPS focuses on the dynamics, which is the evolution of the system state over time, often modeled by differential equations and stochastic processes.

2.1.1 The Design Process

In [27], Lee presented several challenges of CPS design. Compared to general-purpose computing, CPSs have always been held to a higher standard of reliability and predictabil-

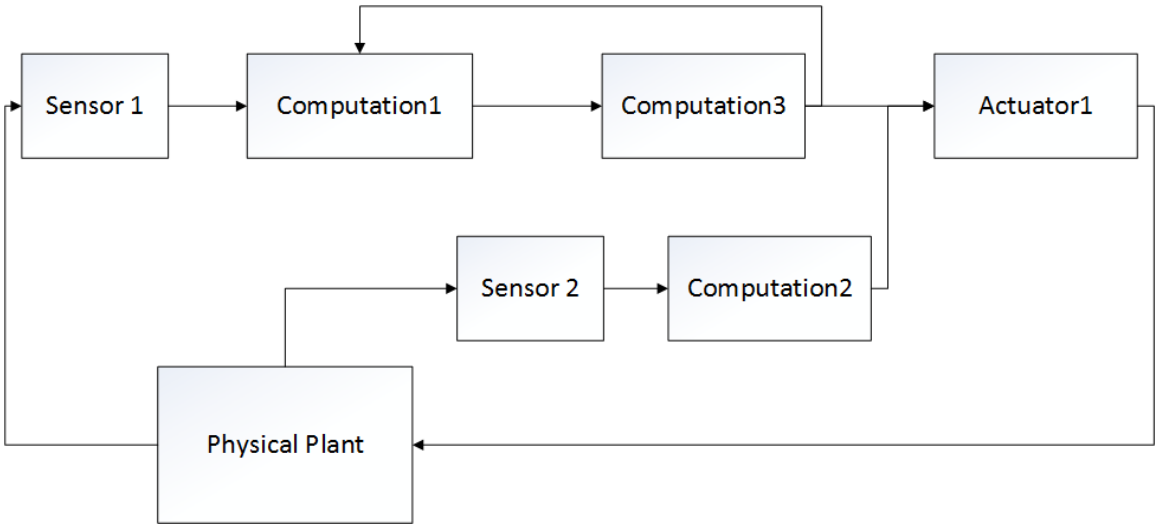


Figure 2.1. Typical Structure of a Cyber-Physical System

ity. Such applications as traffic control, automotive safety and health care would cause disaster without improved reliability and predictability. For that reason, there are requirements that CPSs should satisfy in a controlled environment; they must be resistant to unexpected conditions and adaptable to system failures.

In [18], Lee *et al.* provided a general structure and design process for a typical CPS to handle these challenges. While the main objective of this dissertation is to focus on the *fault-tolerance* aspects of the design, here we summarize the essential part of the process.

Figure 2.1 shows an example of a CPS. This is a typical networked platform with two sensors and three computation components. The action taken by the actuators affects the physical plant, whose state information is then measured by the sensors as inputs to the control computers. In this figure, computation 2 implements a *control law* with the input data from sensor 2. Sensor 1 makes additional measurements of the data and sends messages to computation 1, which cooperates with computation 3 to implement additional tasks via the network fabric between computations 1 and 3. The whole structure uses two loops called the *feedback control* loops. For example, a typical autonomous system such as Google's *self-driving car* [56] would use computation 1 for *localization* algorithms, computation 2

for *motion planning*, and computation 3 for control. Notice that most, if not all, of these tasks are *real-time* tasks, which we will discuss in detail in later sections.

The design process can be divided into three sub-processes: *modeling*, *design*, *analysis*. This methodology is called *model-based design*. Models specify what a system does, normally through mathematical differential equations. Design is the processs of creating the system structure. It specifies how a system does what it should do based on the requirements. Analysis is the process of obtaining a detailed understanding of a system through various techniques such as data analysis, simulation and mathematical proof. It specifies why a system does what it should do [18].

The modeling sub-process mainly consists of *continuous dynamics* modeling, *discrete dynamics* modeling, *hybrid system* modeling including both continuous and discrete systems, and *concurrent models of computation* with which parallel computations would occur [18]. The continuous dynamics modeling uses differential equations to describe the continuous systems (these systems are typically from the physical world). On the other hand, the discrete systems (typically from the cyber side) employ *state machines* for modeling. Hybrid systems combine both techniques, and the concurrent models of computation take advantage of popular *timed models of computation* such as *Petri Nets*. Interested readers can refer to [18][41] for more information.

The design phase begins with selecting hardware and software components (e.g., motors, batteries, sensors, microprocessors, memory systems, operating systems, wireless networks), and then putting them together in an efficient and reliable way [18]. In this dissertation we focus on the system and software architecture design. In particular, we will discuss how to design for fault-tolerance in real-time systems. For comprehensive discussion about hardware and software design, refer to [18][39].

The analysis phase is for analyzing and verification of the design [18]. For example, the system may be analyzed for safety conditions, such as the vehicle speed should be no greater than 0.1 meter/sec, or the system may require timing behavior verification, for

example the air bag system should respond to an emergency within a very short time. Many modern CPSs also require security and privacy verification. In [18] and [39], the authors discussed several popular techniques including but not limited to, *invariant and temporal logic*, *reachability analysis and model checking queuing theory*.

The three design phases are not isolated from each other, but interact. The design of a CPS is an iterative process, with modeling, design and analysis mixed together with each part of the system. For example, before the design process, engineers would be provided system requirements, including requirements of safety, reliability and timing. If the analysis of one part of the system does not satisfy the requirement, one should go back to the previous phase for modifications.

2.2 Fault-Tolerance

This dissertation is about design for fault-tolerance in CPSs. In this section we introduce the state-of-the-art fault-tolerance techniques.

In [10], Burns *et al.* gave a definition of *fault-tolerance*: the system continues functioning in the presence of faults or errors, with no significant functionality or performance losses. In [10][23][40], the authors discussed several types of faults, errors, and failure models of the system. We summarize these as follows.

2.2.1 System Faults, Errors, Failures, Reliability, and Safety

According to [10], a *fault* is a system "bug", hardware or software, that would inevitably happen whether or not the system was designed correctly. An *error* is the result of the fault which is *activated* somehow. An *failure* is in turn the result of the errors if the errors are significant enough to violate the system requirements. The *reliability* of the system is a measure of the success with which the system obeys the authoritative requirements of its behavior. The *safety* requirement of the system is the requirement that the failure should not cause a disaster such as the death of a human being.

There are three types of faults: *transient faults*, *permanent faults* and *intermittent faults*. Many faults in CPSs are transient, which is one of the main assumptions of our research. System errors can be divided into *value errors* and *timing errors*. Value errors refer to erroneous outputs due to software bugs or faulty hardware components. Timing errors refer to the service that is delivered at the wrong time, such as missing the *deadlines* of a real-time system.

System failures can also be divided into different types. According to [40], an *uncontrolled system failure* means that the system has not detected the fault yet, or although having detected the fault, it has not yet invoked a *controlled system failure handler*, possibly resulting in significant system damage and severe safety conditions.

Also from [40], *degraded performance mode* refers to a system that, in presence of faults, operates without significant performance or safety level loss. *Fail safe mode* refers to a system that would enter a safe condition after detecting faults, with some performance degradation. An example would be a railway signaling system, which turns all of its lights to red after detecting faults. *Fail silent mode* refer to a system that enters into a pre-determined static mode, such as rebooting or shutting off the system, so that it will cause little or no damage.

It is also worth noting that several international safety standards and guidelines have been developed for many *safety critical systems* [40]. For example IEC 61508 refers to generic industrial standards, ISO 26262 refers to automotive systems standards, etc. For developers using C, the "MISRA C" guidelines are widely employed as a safety "language subset" [1].

2.2.2 Mean Time to Failure

We now show how to calculate the system reliability and MTTF. Let T denote the lifetime of a component, $f(t)$, and $F(t)$ the probability density function and the cumulative

distribution of T, respectively. $F(t) = Prob\{T \leq t\}$ is the probability that the component will fail at or before time t, and the density function $f(t)$ is given by [23]:

$$f(t) = \lambda e^{\lambda t} \tag{2.1}$$

where λ is called *failure rate*. Then $F(t)$ can be derived as:

$$F(t) = 1 - e^{-\lambda t} \tag{2.2}$$

$R(t)$, the reliability of a component (the probability that it will survive at least until time t), is given by

$$R(t) = Prob\{T > t\} = 1 - F(t) = e^{-\lambda t} \tag{2.3}$$

Therefore, a constant failure rate indicates that the lifetime of the component has an exponential distribution with a constant λ. Then, MTTF, the expected lifetime of a component, is given by:

$$MTTF = E[T] = \int_0^\infty t f(t) dt \tag{2.4}$$

Substituting $\frac{dR(t)}{dt} = -f(t)$ yields:

$$MTTF = \int_0^\infty R(t) dt \tag{2.5}$$

For the case of a constant failure rate for which $R(t) = e^{-\lambda t}$,

$$MTTF = \int_0^\infty e^{-\lambda t} dt = \frac{1}{\lambda} \tag{2.6}$$

For a detailed derivation of these equations, refer to [23].

2.2.3 Current Techniques for Fault-Tolerance

In [10], Burns *et al.* gave definitions for different levels of fault-tolerance. *Full fault-tolerance* means the system requires full masking of faults in order to perform most, if not all, of its functionality. *Graceful degradation* or fail soft, means the system continues to operate in the presence of errors, accepting a partial degradation of performance. In this dissertation we define several different levels of fault-tolerance, but the main ideas are similar to the above definitions, as will be discussed later.

All fault-tolerance techniques take advantage of extra redundant components introduced into the system [10][23]. Traditional fault-tolerance techniques for hardware is via continuous massive deployments with *duplex*, *triplex* or *M-of-N system*. Duplex refers to two copies of the software tasks running on two processors, which can detect but not mask out faults. Triplex refers to three copies of the tasks running on three processors, which can mask out 1 erroneous output. M-of-N system is the generalized version of triplex, which uses N copies of the tasks running, and can mask out N-M erroneous outputs. Software fault-tolerance includes *acceptance tests*, *N-version programming* (which uses N different versions of the software tasks), *recovery block approach* (which uses a backup version once a fault is detected), *remote procedure calls*, and *check pointing* (which rolls back to previous time snapshot once a fault is detected). Readers may refer to [10][23] for more details. In this dissertation, we will use an adaptive fault-tolerance technique that is able to adjust the runtime deployment and select the most suitable versions of the tasks, based on the safety condition of the physical plant.

2.3 Real-time System

Most of CPSs are *real-time systems*, where the outputs of the software control tasks should be delivered before a deadline. The reason is that typically the input corresponds to some physical event such as a change in the speed of a ground vehicle, and the output has to respond to that change, for example, by invoking the braking or accelerating commands

to adjust the vehicle speed. The delay from the input time to the output time must be sufficiently small within some allowable time frame [10]. The field of real-time system is extensive, and we will only summarize what is relevant to our research.

From [10], *hard* real-time systems refer to those systems which require completion of certain responses within the specified deadlines. *Soft* real-time systems refer to those systems where response times are important but the systems will continue to operate correctly even with occasional deadline misses. *Mixed criticality* systems are those systems where the tasks can be classified according to different critical levels [11]. The higher the task criticality level, the lower the failure rate it should have. In all of the above three systems, the failure covers both value and timing errors. Many real-time systems today may have hundreds of tasks, with only a sub-set of which are critical. Our main focus is to optimize the expected lifetime and the long-term reliability of the processors running these critical tasks.

There are two approaches to the design of software for a real-time system: a *time triggered* (TT) architecture and an *event triggered* (ET) architecture [40]. Implementation of a TT architecture will typically involve the use of a single interrupt, which is linked to the periodic overflow of a hardware timer [40]. The interrupt will be used to drive a software program called the *interrupt service routine* (ISR), which in this case will be a task *scheduler* (a simple form of the *real-time operating system* or RTOS). The scheduler will, in turn, release the system tasks (software tasks in real-time systems) at the predetermined time points [40]. Implementation of an ET architecture will typically involve the use of multiple interrupts. Each interrupt is linked with particular periodic and aperiodic events. Standard RTOS will be used for the design of ET systems.

Real-time scheduling is an essential part of a real-time system. In the TT architecture, the scheduler often deals with a ”co-operative” task list, within which each task must be completed before another task can be executed; there is also provision for priority interrupt, where higher priority tasks could *preempt* other tasks. Therefore, this scheduler

has the highest possible determinism, which is ideal for systems where accurate execution times and minimum *task jitters* are required [40]. In contrast, the ET architecture with RTOS has much lower determinism (especially when handling task jitters), however it has more flexibility to handle the *sporadic* or aperiotic tasks. The main scheduling algorithm used in industry today is the *rate monotonic* (RM) scheduling, in which the tasks with shorter periods have higher priorities [15][28]. There is extensive research in the real-time scheduling theory, but very few are actually used in industry. Some of the popular scheduling algorithms in academic research include *earliest deadline first* (EDF) [52], *fault tolerant scheduling* [22][31], *power aware scheduling* [2]. Readers can refer to [15] for a comprehensive survey of real-time scheduling algorithms. In this dissertation, we will use the ET architecture with an RTOS and rate monotonic scheduling for each processor.

2.4 Machine Learning for CPSs

Traditional simulation-driven approaches for the research and the development of CPSs are gradually being replaced by the modern data-driven approaches. Data collections can be handled either from the real hardware, or from realistic simulators.

Machine learning therefore became a core technology for many CPS designs, especially for robotics systems that must capture large amounts of data. For example, learning has become a core part of the research in computer vision. Data-driven techniques are being used to tackle problems for which it is very difficult to develop well-formatted closed form mathematical equations. The goal is to use the ever-growing amount of data from the sensory sources to *train* a particular pattern that can be used to make predictions from the unseen data.

In this dissertation, we use many machine learning techniques, including the regressions, classifications and the reinforcement learning for tasks such as the predictions of the *quality of control* (QoC), reliability, fault-tolerance level, and the optimal real-time task load.

CHAPTER 3

TECHNICAL FUNDAMENTALS OF ADAFT

In this chapter, we will describe the problem formulation and certain important technical fundamentals of our framework.

3.1 Problem Formulation

Our framework aims to achieve computing resource efficiency in CPSs. In traditional fault-tolerance techniques, massive redundancy such as triplex, is deployed all the time for tolerating hardware faults [23]. Multiple versions of software real-time tasks with a switching mechanism such as *Simplex*, are used for software fault-tolerance [44]. Our framework, AdaFT, attempts to minimize the actual runtime deployment of redundancy as much as possible, while keeping the same level of system safety.

While satisfying the safety requirement, AdaFT will reduce the level of runtime deployment of fault tolerance and still improve the long term system reliability. We use a *thermal aging acceleration factor* (TAAF) to represent the thermal reliability of computer systems.

Most CPSs have both safety and quality of control/service (QoC) requirements. For example, for a highway platoon system, the safety requirement is that the inter-vehicle spacing should not be less than a threshold, while the QoC requirement is that the average inter-vehicle spacing should be kept reasonably small. The safety requirements must be satisfied, while the QoC requirements are soft constraints that should be achieved as much as possible. These requirements can be satisfied using *optimal control* methods, such as *linear quadratic regulator* (LQR) or *model predictive control* (MPC) [59]. Together with

the TAAF, AdaFT attempts to find an optimal real-time task loading during runtime to satisfy both safety and QoC constraints, and optimize TAAF.

3.2 Technical Background

3.2.1 Failures in Cyber-Physical Systems

It has long been understood that failures in CPSs can be treated in a more application-specific way than failures in general-purpose systems. Meyer's notion of *performability* specifies accomplishment levels for the controlled plant and calculates the probability that the real-time controller will function well enough to meet each of these accomplishment levels [21][33][34]. Another approach relies on the fact that the controller is in the feedback loop of the controlled plant [26][47]. Therefore, any computational delay contributes to the feedback delay. The impact of feedback delay on the controlled plant performance is well understood in control theory. By quantifying this impact on the controlled plant performance, one can obtain *cost functions* to express the degradation of the quality of control. Obviously, the state of the controlled plant affects the impact of feedback delay on the quality of control.

Note that this approach presupposes the existence of sufficiently accurate models of the controlled plant. Such models would be needed, in any case, to assess the effectiveness of any control algorithm used.

3.2.2 A State-Space Approach

In contrast to general-purpose systems, cyber-side failures in a CPS can be defined in an application-focused way. This happens when the controller is unable to keep the controlled plant within a designated subset of its state-space, called the *Safe State Space*, S^3. Full details can be found in [24].

S^3 is defined as follows [24][25]: based on the application, the user or application engineer (or other application-domain specialist) can specify the constraints that the plant must

satisfy in order to be considered to be operating safely. These are called the *Safety Space Constraints* (SSC). For example, the maximum allowed G-force on an aircraft, together with the aircraft dynamics, can be used to specify constraints on the pitch, yaw and roll as well as the rate of change of these variables.

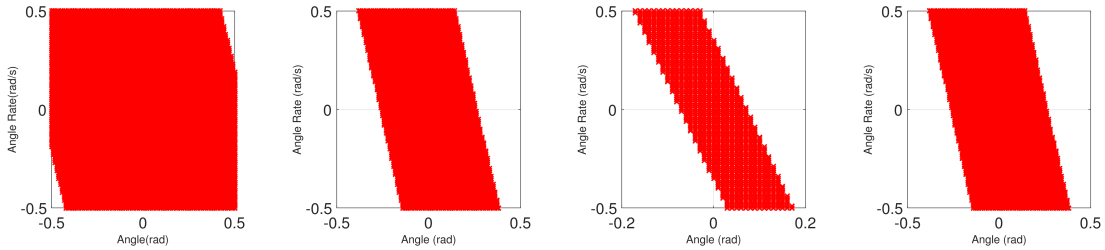
A point is in S^3 if (a) the plant satisfies the SSCs at the present time and, (b) based on the plant control laws, the control algorithm used, the actuator limitations, the control task execution policy and rates, and the specified limits of the operating environment impact on the plant, the plant will continue to satisfy these constraints up to a given horizon, as long as the correct control inputs are applied.

The impact of an erroneous controller output on the plant performance depends on the current plant state. If the plant is deep within its safe region of the state space, it may well be able to withstand a certain number of erroneous inputs without impairing safety. Such application specific error-tolerance translates to a lowered requirement for controller fault-tolerance.

As an illustration, consider the canonical inverted pendulum control system, widely used to illustrate CPS concepts. The objective of the control system is to keep the pendulum close to vertical by applying a force to the motorized cart that the pendulum is attached to. The state variables are the angle of the pendulum with respect to the vertical line and the rate at which this angle changes ($\theta(t), \dot{\theta}(t)$). Suppose we define SSC as the region where $-0.5 \leq \theta(t) \leq 0.5$ radians.

If zero-order control is provided [59], every period of P seconds, a new control input is calculated and applied. This control is kept steady throughout the period until a new control input has been calculated. We now examine the impact of two parameters on S^3: the period and the maximum control force that can be applied.

Figure 3.1 shows S^3 for two different control periods and two different control force bounds for a unconstrained maximum control force denoted by u_{lim}. Note that even though the SSC is the same for all cases, the size of S^3 shrinks as the period increases, indicating



(a) Period = 10ms (b) Period = 30ms (c) u_{lim} = 2.5 N (d) $u_{lim} > 10$ N

Figure 3.1. Safe State Space (S^3) for the Inverted Pendulum System with Different Control Periods and Control Force Bounds ((a)-(d))

the increasing vulnerability of the plant due to a reduction in control update frequency. As the control period increases, the real-time control tasks are executed less frequently, leading to a potentially worse quality of control. Also, note how S^3 shrinks as the control capability decreases. Increasing the range of control forces that can be applied is not always a good idea: a decrease in control capability may sometimes *reduce* the plant vulnerability to failure [24].

3.2.3 Adaptive Fault-Tolerance

We will now show how the state-space approach leads naturally to adaptive fault-tolerance. We define three sub-spaces within S^3 as follows:

S_1**:** No fault-tolerance is required. The controlled plant is in a region of the state-space where even if the actuators are held at their worst-case incorrect setting until the next iteration of the control task, the plant will not leave its S^3. Hence, only one copy of the control task needs to be executed. Even if the task fails and produces the worst possible incorrect control output value, the plant remains safe and can be recovered in later periods.

S_2**:** It is sufficient for the controller to be fail-stop, i.e., the system generates only two types of controller output: correct or default (e.g., zero) output. Only error detection rather than error correction is needed in the control output calculation. For instance, one could use a processor duplex with two independent control calculations being compared.

If a significant mismatch (i.e., outside the range of numerical approximations) is detected between the two outputs, then an error is declared in the computation and a zero control input (or other default value) can be applied.

S_3**:** Full fault-masking is required. If the controller produces an incorrect output, the plant cannot be guaranteed to stay in the safe state-space. Therefore full-strength fault-tolerance with fault-masking should be used, e.g., a triplex with majority voting [23].

Hence, if a plant is in S_1, no fault-tolerance is needed; only one copy of the control task needs to be executed. Even if the task fails and produces the worst possible incorrect control input value, the plant remains safe and can be recovered in later periods. If it is in S_2, only error detection rather than error correction is needed in the control input calculation. Finally, if the state is in S_3, then the full-strength fault-tolerance with fault-masking should be used, e.g., a triplex with majority voting [23]. Note that all other states outside of S_1, S_2 and S_3, i.e., outside of S^3, are either physically unachievable, or uncontrollable even by a perfect controller. The latter means that even if an always correct control input is applied, the physical plant might still enter the unsafe region violating the SSC. In such a case, it still needs the full level of fault-tolerance, but it is not guaranteed to be always safe.

With these sub-spaces, a *state-based adaptive fault-tolerance* can be developed. Since the controlled plant is in S_1 for most of the time, a lower level of fault-tolerance can be used, to reduce the amount of stress on the controller or to use the available released computational capacity for other tasks [24].

3.2.4 Impact on Thermal Age Acceleration

It is well known that the workload affects processor reliability. With a higher workload, the thermally-induced failure rate increases [24][36][45][46]. Operating at higher temperatures accelerates the device aging process. The rate at which such aging occurs can be captured by means of the *Thermal Age Acceleration Factor* (TAAF). If the TAAF over some time interval δt is α, the effective aging of the circuit over that interval is $\alpha\delta t$. α is a

strongly increasing, non-linear function of temperature. The reliability advantage of adaptive fault-tolerance is that its lower computational burden reduces the average operating temperature and hence the amount of circuit aging.

3.3 Related Work

The idea of adaptive fault-tolerance is not new; it goes back more than two decades. In [20], the authors pointed out that the fault tolerance needs of the application, and the fault-tolerance capabilities of the micro-controller could change as time goes by, therefore an adaptive technique was presented. In [19], the authors proposed an object oriented way to manage adaptive fault-tolerance. The fault tolerance management unit is informed about the reliability requirements of the application; it then adjusts the level of fault-tolerance to suit the reliability requirements.

Considerable recent work has also contributed to the area of adaptive fault-tolerance, due to the fact that increasingly more safety critical systems are nowadays controlled by embedded controllers. In [32], an On-demand Real-TimEGuArd (ORTEGA) was proposed to allow for efficient resource utilization based on the runtime state space. The idea is to divide the state space of the controlled plant into two subsystems: a high-assurance-control (HAC) and a high-performance-control (HPC) subsystem. They introduced a two-level FT option to be adjusted in real-time.

Most real-time, fault-tolerance research assumes that faults cause failures at the application level. In [51], Song, *et al*. explained why system-level software (e.g., RTOS scheduling, memory management and I/O processing) is closely tied to failures; indeed, 65% of hardware errors would corrupt OS state [29]. They claimed that such a situation has received little attention, and proposed Computational Crash Cart (C^3), with the main idea of dividing the system components based on their functionality (i.e., scheduler, I/O). After a fault is detected, the faulty component can perform a focused micro-reboot, rather

than having to restart the whole system. Later they extended this work in [50], to allow the system to have a runtime monitoring as well as validation capability.

Swetha, *et al* introduced the *Enhanced Resource Management Scheme* (ERMS) and compared it with the traditional redundancy approach [55]. They introduced a new scheduling method, which combines the dynamic planning and dynamic best effort approach. This approach allows both periodic and aperiodic tasks, as well as an online reconfiguration for error management. During operation under this fault recovery technique, all critical tasks will meet their deadlines, and the system can still function at a reduced but above minimal safe functionality in the presence of errors. This scheme has been analyzed and evaluated, using simulation, on an automotive cruise control system.

Bak, *et al* developed a combined online/offline approach [3] for the well known Simplex architecture. Their approach uses aspects of a real-time reachability computation, which also maintains safety, but is less conservative. The switch logic for their Simplex architecture will, like the traditional Simplex architecture, guarantee that the system never enters an unsafe state, but uses the complex controller as much as possible. Simplex deals with software fault-tolerance, with the assumption that the simple controller is formally verified and is free of software bugs, and the complex controller provides potentially better quality of control but cannot be formally verified. Since only one copy of each controller version is deployed during runtime, no hardware fault-tolerance is provided. One possible combination of our work and the Simplex architecture, is to introduce a more flexible and more powerful fault-tolerance hierarchy (both software and hardware): with the knowledge of the physical side state information, the system might be able to determine which version of the control task and how many copies of the control tasks to run at a specific time.

The paper by Bogdan and Marculescu [6] argues that one can expect many cyber-physical computational workloads to show fractal behavior. They point out that if this is the case, it will affect computational resource allocation. As far as AdaFT is concerned, what is of relevance is the computational burden as a function of the application (controlled

plant) physical states. In its current form, the framework accepts traditional digital control workloads. As new workload formulations emerge from the CPS community, they can be included in the AdaFT framework.

Note that most prior work on adaptive fault-tolerance focused on the cyber side of CPS. The cyber system was either informed about the current state by the physical plant and the required fault-tolerance level, or these conditions were assumed to be given. By contrast, our work uses a cyber and plant side co-design approach to provide in real-time the physical side state information to the cyber side, based on which the cyber fault-tolerance level can be determined.

CHAPTER 4

IMPLEMENTATIONS OF ADAFT

4.1 Structure of the AdaFT Framework

The structure of AdaFT is shown in Figure 4.1. It consists of two major parts: sub-space determination and analysis. The first part focuses on the generation of the sub-spaces, and a machine learning approach for sub-spaces classification; whereas the analysis part uses the outputs from the sub-space classifier, and simulates the system to perform reliability analysis. AdaFT takes the physical side information of the controlled plant as input, implements an adaptive fault-tolerance approach to guarantee the same safety level as the traditional approach would do, while keeping the computing resource usage as low as possible, as well as improving the long-term reliability of the computing platform.

4.1.1 Sub-space Determination Component

The sub-space determination component consists of two parts: *sub-spaces generation*, and *sub-spaces classification*. The *system model* is a mathematical description of the physical system, typically given as a set of differential equations. In this work we assume the system model can accurately represent the real physical plant, since the major implementation is based on simulation. As one of the inputs of the sub-space generator, AdaFT actually uses a *reduced order system model*, which will simplify the system model to a reasonable level while still maintaining the key physical components. This reduced order model has the potential of significantly reducing the total amount of time collecting data for the sub-spaces. We will discuss how to obtain this model in later sections.

Control tasks are the real-time control tasks that control one or more of the physical state components. We focus here on periodic tasks; sporadic tasks with the period replaced

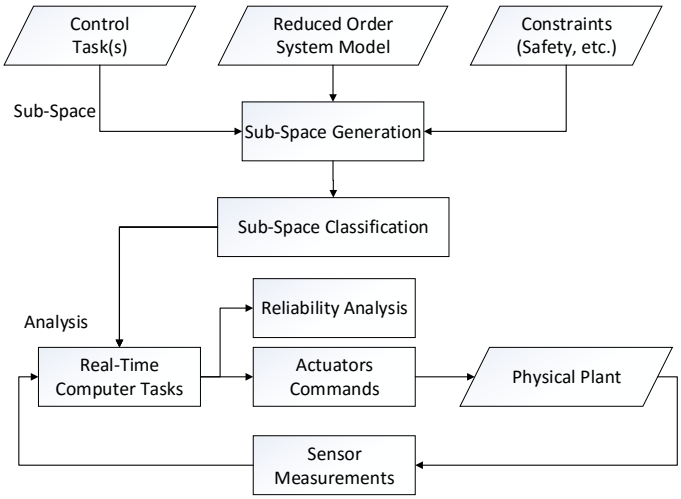


Figure 4.1. Software Architecture of AdaFT

by a minimum invocation between successive tasks can be incorporated easily. Typically, each task will have a task period, deadline, a *worst case execution time* (WCET), and power consumption. These parameters are usually determined during the design time and hence are known in advance.

Constraints are the safety space constraints SSC, such as the minimum inter-vehicle spacing for *adaptive cruise control* (ACC) system on highways, or the allowed angle range for the various joints of a robot.

Sub-space generation is one of the core parts of AdaFT, which divides the operating space of the CPS into S_1, S_2, and S_3 sub-spaces defined in Section 3.2.3. These sub-spaces will in turn determine the level of runtime deployment of the fault-tolerance needed to ensure the system safety.

Sub-spaces classification takes as inputs points from the sub-spaces, and then uses machine learning techniques for their classifications. The purpose of this part is to efficiently compute the FT level during runtime, as the total size of the three sub-spaces are typically

very large. Most machine learning algorithms only need small memory to store the fitted model for real-time classification, and the running time for these are in the order of milliseconds, or even microseconds [38].

4.1.2 Analysis Component

The analysis component of AdaFT uses a *real-time computer* model, with the support of *real-time operation system* (RTOS) and *real-time scheduling* policy to execute the control tasks. The level of software and hardware redundancy depends on the sub-spaces and the corresponding trained parameters. *Reliability* model should be included for analysis. Our current implementation uses the TAAF metric. Finally, the *physical plant* model uses realistic simulators, such as *Carsim* [48], for the simulations.

4.2 Implementations

We have implemented AdaFT in both Python and Matlab. We chose Matlab due to its popularity in many engineering domains. Python is gaining popularity due to its powerful numerical and machine learning libraries such as *numpy*, *scipy* and *sklearn*. In this section we sketch several of the technically interesting aspects of the implementation.

4.2.1 Sub-Space Generation

The sub-space generator is a core component of AdaFT. It takes as inputs the control tasks, the system dynamics model, and the SSC safety constraints. Then, for each given state in the entire state space, AdaFT determines whether it is in S^3 by simulating the system to a certain time horizon or final condition. The three sub-spaces, S_1, S_2 and S_3 are then generated from S^3.

Algorithm 4 shows how to generate S^3 through simulation based on the system dynamics. Each intermediate state before the time horizon (or final condition) needs to be checked against SSC. If all are within SSC, then this specific initial state is within S^3.

ALGORITHM 1: S^3 Generation

Input: The operational state space
Output: S^3;
for *All x that satisfy the SSC* **do**
    run simulation with x as initial condition, with control period (step length) δt and correct control, until the time horizon is reached or the final condition is satisfied;
    if *all intermediate states satisfy the SSC* **then**
        S^3.add(x);
    end
end

Algorithm 2 shows how to generate S_1, S_2 and S_3. It takes S^3 as input, and for each state x in S^3, it simulates the controlled plant for one task period. For S_1, the worst case wrong control is applied. If after one task period, the physical state of the plant is still within S^3, this state x is within S_1. On the other hand, if the plant is in S^3 with zero control after one control period, x belongs to S_2. Finally, based on the definition of S^3 and S_3, x is in S_3 if it is in neither S_1 nor S_2. If there are multiple tasks in the system, each individual task will have its own set of sub-spaces. When generating the sub-spaces for one particular task, we assume other tasks are producing correct outputs. Note that there are almost always infinitely many points in S^3, therefore, one might need random sampling to be applied, or other techniques (for example, by changing the simulation granularity), to choose x so that the software will finish in a finite amount of time. We will discuss this complexity issue in later sections.

Remark 1: It should be noted that both hardware and software fault-tolerance can be applied using AdaFT. We have already discussed hardware fault-tolerance using duplex and triplex. Software fault-tolerance techniques are similar in the sense that they use redundancy. Examples include *N-version programming* which is a static redundancy technique [23]. There are also error detection and recovery techniques such as the *recovery block approach* [23]. All AdaFT needs to know is what sub-space the physical plant is currently in. If the plant is in S_1, then no FT is needed (neither hardware or software);for S_2, only duplex for hardware FT or error detection such as 2-version software is sufficient followed by a 0

ALGORITHM 2: sub-spaces Generation

```
Input: S^3;
Output: S_1, S_2, S_3;
for All x in S^3 do
    run simulation with x as initial state with control period δt with the worst case
    wrong control;
    if after one period the state is within S^3 then
        S_1.add(x);
    else
        run the same simulation with zero control;
        if after one period the state is within S^3 then
            S_2.add(x);
        else
            S_3.add(x);
        end
    end
end
```

control input for a fail stop model; and finally for S_3, triplex for hardware FT or 3-version programming for software FT may be used. The user can decide whether to use hardware only, or hardware and software fault-tolerance together. A real-world example is the Boeing 777 flight control system. A single Ada program was produced but three processors (hardware triplex) each using a distinct compiler (software 3-version dynamics) were used for fault-tolerance [10]. For such a case, S_1 corresponds to only using one processor with one compiler, S_2 corresponds to using two processors with two different compilers, while S_3 is the current default option.

Remark 2: The complexity of this approach is proportional to the number of voxels of the state space that are evaluated. This number is obviously exponential in the number of state space dimensions. However, we do not require that the entire safe state space be evaluated. Each voxel in S^3 starts, by default, in S_3; it may be reclassified as in S_1 or S_2 following an evaluation. For this approach to be useful, it is sufficient to evaluate the more frequently visited parts of the state space, which can be obtained by gathering traces of the state space trajectory and evaluating the state space neighborhoods of these points.

Remark 3: Note that we do not explicitly model communication faults. Highly effective coding and other redundancy mechanisms exist to reduce communication errors to desired levels. If necessary, the event of an undetected/uncorrected error in communication can be included in the failure probability of the relevant task.

4.2.2 Worst Case Controller/Actuator Action

Recall from the definition of S_1 as a sub-space where the application of *any* physically realizable input over one control period will not cause the system to leave the safe state space. The following approach can be used to generate S_1.

The user has to specify a cost function indicating divergence from the desired state trajectory or value. Control is usually applied to ensure minimization of such a function. However, one can instead try to maximize such a function given the control input constraints. If such a maximum divergence still keeps the controlled plant within S^3, that point will be declared to be within S_1. A good rule of thumb for this type of cost function is to use $cost = \sum_i (\frac{x_i - x_{id}}{x_{id}})^2$, where x_i is the actual values for the ith state variable which we care about its safety, and x_{id} is the corresponding desired value. Note that the *feature scaling* [38] of the state variables is important (i.e., normalizing each variable into the scale of 0 to 1), since the total cost will otherwise mainly consist of the states with large absolute values. Further details can be found in [38].

This approach to compute the worst case control is essentially a simplified optimization problem. The only constraints for this problem are the control input bounds, normally provided by the specifications of the actuators. Algorithms to solve optimization problems with potentially complicated state dynamics might become computationally expensive, however, since this step is executed offline using simulation software, execution time will not be an issue.

It should be noted that for many applications it is sufficient to use some heuristics to determine the worst case control/actuator outputs. For example, in the ABS braking system,

the controller will steer the *slip ratio* towards its optimal point (a typical value is 0.2). If the current slip ratio is greater than this value, then the worst case actuator command is equal to the upper bound of the actuator value with the opposite control direction.

4.2.3 Multiple Control Tasks

The approach introduced in Section 4.2.1 can be generalized to control plants with multiple control inputs. At runtime, AdaFT will determine the number of copies for each task at each iteration release. The approach to determine the subspace of S_1, S_2 and S_3 runs as follows.

First, we need to decide which state points belong to S^3. This step is the same as before, i.e., the correct control inputs for each task are applied to the plant with a particular initial state condition. If, to a time horizon, all intermediate states satisfy the SSC, this initial state vector belongs to S^3. Note that to speed up the process, all of the intermediate states should also be added to S^3, since the final condition for the initial state is the same as the one for all those intermediate states.

Next, in order to determine S_1, we need to determine the worst control inputs vector for all tasks, given a specific initial state condition. We apply this worst case control inputs vector to the plant for one control period for a particular task. Notice that each control task may have a different period, therefore the time to apply this worst control vector might also differ depending on which control task has been released.

The way to find a worst control inputs vector is identical to the single control task case. First we define a cost function regarding the safety requirements of the plant, $cost = \sum_i (\frac{x_i - x_{id}}{x_{id}})^2$. Then, several constraints can be defined, such as the actuator bound, and the same optimization approach can be applied, where the decision variables in this case are the control inputs for all tasks.

For S_2, the procedure is similar to S_1: we apply a zero or default control (e.g. control from last period) for the tasks under consideration and apply the worst control for all the

other tasks. After one control period of a particular task, if the plant state vector still remains inside S^3, this initial state belongs to S_2. Note that the worst control for all the other tasks are applied here, since we don't have control over the inputs to the other tasks; they can be any value in case of failures.

It should be noted that for multiple control tasks, each task would have a corresponding subspace that consists of S_1, S_2 and S_3. Now let us take an example of 3 tasks: t_1, t_2 and t_3, with the period of 10 ms, 10 ms, and 20 ms, respectively. After obtaining S^3 with the correct control, the next step is to obtain S_1. For S_1 with respect to t_1 and t_2, we need to use the worst control vector to run the simulation for 10 ms, and check if the plant is still within S^3 at the end of that period. While for S_1 with respect to t_3, we run for 20 ms and then check. In order to obtain S_2 for t_1, we use zero control for t_1 and the worst control vector for t_2 and t_3 and simulate for 10 ms. If the final state is still within S^3, that point belongs to S_2. S_2 for t_2 and t_3 can be obtained using the same procedure.

4.2.4 Reduced Order System Model

For simpler controlled plants it is feasible to use a full order system model to generate the sub-spaces. For more complex plants, we use machine learning techniques, for example, *feature selection*, and *precision-recall* trade-off [38]. We first sample certain amount of data with a coarse granularity, run simulations, and classify these samples as to whether or not they violate the SSC. We split the resulting data set into training and testing sets. The training set is for fitting a particular machine learning model, while the testing set is for testing the accuracy of the model for unseen data using the fitted model [38]. The next step is to use a machine learning algorithm that can calculate he importance of each feature, e.g., *random forest*, to fit the data. At this point, if the testing accuracy is higher than a threshold, (e.g., higher than 98%), and the *precision*, i.e., the percentage of the correct prediction of the class belonging to S^3, is also high enough, (e.g., higher than 99%), we can assume the fitted machine learning model can not only fit the training data very well,

but can also be well generalized. We can then use this simplified fitted model, rather than the real simulations which are more time consuming and complex, to generate more data for S^3.

On the other hand, if we observe a high training accuracy with a lower testing accuracy, this typically means an overfitting of the model, or *high variance*. We must either reduce the model complexity or get more data. For the first approach, we can remove the features with relatively low importance from the model. For the latter, we can sample more data with respect to the more important features, again according to the feature importance. If there is a low training accuracy, which means an underfitting or *high bias*, we must first fit the training set using more advanced techniques such as the *ensemble* model that combines several weaker models to achieve a better one. Our results show that most CPSs have a high training accuracy due to the clear relationship between the inputs (the previous state values and the control inputs) and the outputs (the new state values), determined by differential equations. In all of our case studies, we achieved very high training and testing accuracy. If, for any case, it is impossible to achieve high accuracy, we need to sacrifice in terms of *recall* which is the proportion of true positives (data points belonging to S^3) that are correctly predicted as such, for high *precision*. With a slightly lower recall, the system might waste some computing resources for providing unnecessary redundancy; but with a low precision, a hazardous behavior may occur.

It should be noted that other approaches can be used along with machine learning. For example, since each simulation is independent of the other, parallel computing can be employed to accelerate the data collecting process using multiple computers. Further, pruning techniques like *Branch-and-Bound* can be used [14].

We now consider the inverted pendulum [59] as an example to demonstrate the machine learning approach. The system consists of an inverted pendulum mounted on a motorized cart and the pendulum is kept close to vertical by controlling the cart speed. The system has four states: (cart position, cart velocity, pendulum angle, and pendulum angular rate).

We first generate data with the following granularity: the step size of the cart position, cart velocity, and pendulum angular rate are all set to 2 meters, m/s, rad/s, respectively, while the step size of the pendulum angle is 0.1 radians. We define a reasonable operating range for each state component, e.g., (-10, 10), (-10, 10), (-0.5, 0.5), (-10, 10), respectively. For example, we generate data points for pendulum angles from -0.5 to 0.5 radians, with a step size of 0.1 radians. There would be around 10000 data points, which are then simulated, and labeled as 1 if a particular data point never violates the safety constraint (the angle stays in the range of -0.5 to 0.5 radians), and 0 otherwise.

We use 20% of the data as the testing set, and use the *random forest* technique with *k-fold cross validation* to fit the remaining 80% of the data. During the process, we use a grid search to tune the hyper-parameters of the random forest, i.e., the number of trees/estimators.

For the initial data set we have achieved a 99% accuracy for the training set and a 95% for the testing set. The training accuracy is higher than the testing accuracy indicating overfitting. We need to either reduce the complexity or get more data. The feature importance for the four state variables was: 0.285, 0.301, 0.155, and 0.258, from which we can conclude that all of the four variables are equally significant for the classification. Therefore, we followed the second approach and added more data. We decreased the granularity of the cart velocity from 2 to 1 m/s, since it is the most important feature for this particular problem. With the new 20000 data points, we obtained a training accuracy of 100%, a testing accuracy of 99.7% and a testing precision of 99.5%. The random forest technique achieved a good training and testing accuracy, as well as a high precision so we then used it to generate data for S^3.

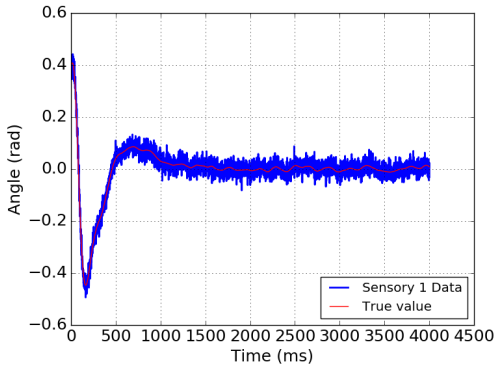
4.2.5 Actuator Noise, Sensor Noise and Failures

Controlled plants are subject to noise or uncertainties from the operating environment. For actuator noise, AdaFT considers the worst case scenario when generating the sub-

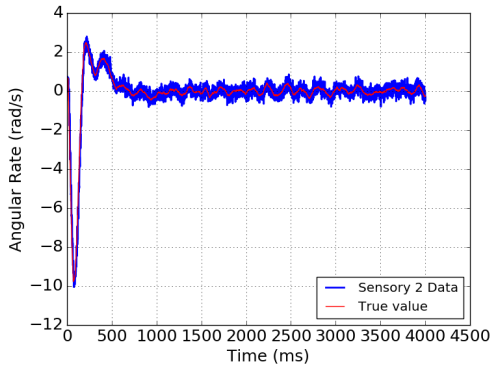
spaces. In AdaFT, the motion model of the controlled plant includes a *state transition probability* $p(x_t|u_t, x_{t-1})$, where x_t, x_{t-1} are the state values at time t and $t-1$, respectively, and u_t is the control input at time t. These probability distributions are derived using the input models of the noise. For a particular final condition provided by the simulation, AdaFT checks, up to a specified confidence interval, whether all states are safe, and if so, the initial state is declared to belong to the corresponding sub-space.

As for the sensor noise, there are many well studied techniques for noise filtering, among which the *Kalman filter* and the *Particle filter* are commonly used in control applications, such as self-driving cars and UAVs [56]. Both techniques are *dynamic Bayesian networks* (DBN). The Kalman filter is an exact tracking algorithm, while the Particle filter is an approximate one. Both use the system dynamics and the control inputs to generate a prior belief about the physical states. This is called the *prediction step*. Then, they calculate the likelihood of the sensor measurements given the initial prior belief. Finally, a posterior belief distribution is obtained for the updated estimation. This is called the *update step*. The Kalman filter produces optimal estimates for unimodal linear systems with Gaussian noise. It calculates a *Kalman gain* which will be used during the update step.

In contrast, the Particle filter uses *Monte Carlo* sampling to randomly generate particles, each corresponding to an initial guess. Then, during the prediction step, it moves the particles based on the dynamics model to obtain the next state of each particle. At the update step, the Particle filter updates the *weight* of each particle based on the sensor reading, which is essentially the likelihood of the sensor reading for each particle. Particles that closely match the readings are weighted higher than those which do not match well. Finally, the Particle filter uses a *resampling* technique to discard highly improbable particles and replaces them with copies of the more probable ones, in order to get the posterior belief distributions. The Particle filter works well for nonlinear systems, whereas the Kalman filter must first perform linearization which might be difficult for some systems. The detailed mathematical derivations of these algorithms can be found in [56].



(a) Angle Tracking(sensor 1)

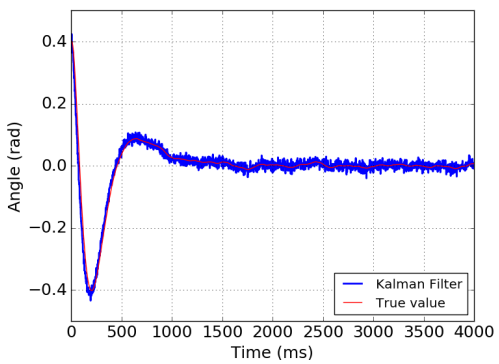


(b) Angular Rate Tracking (sensor 2)

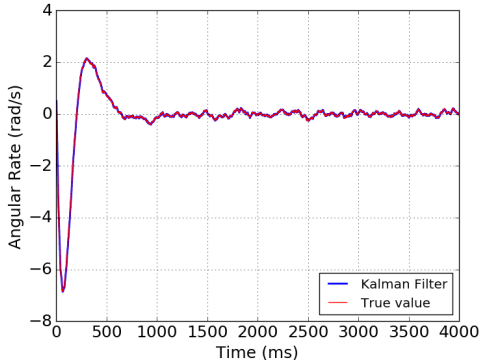
Figure 4.2. Raw Sensory Data ((a)-(b))

AdaFT uses the Kalman filter to track linear systems, and the Particle filter for non-linear systems. We used the Kalman filter for the inverted pendulum mentioned earlier. The initial state conditions are set to (0, 0, 0.4, 0.5) and the actuator noise standard deviation to 0.06 N. We used two sensors with different profiles for the tracking and assume that both sensors have the ability to measure angle and angular rate. The two sensors have an angle noise standard deviation of 0.01 rad and 0.002 rad, respectively, and have an angular rate noise standard deviation of 0.005 rad/s and 0.1 rad/s, respectively. We assume that sensor 1 is better at sensing angular rate, while sensor 2 is better at sensing angles. Figures 4.3 - 4.5 show how filtering algorithms can reduce the sensor noise, and improve the tracking accuracy and confidence.

In order to handle transient and persistent sensor failures, AdaFT extends the standard Kalman and Particle filter algorithms. As discussed before, during the update step, these filtering algorithms calculate the likelihood of the sensor measurements given the prior beliefs. For Particle filters, the calculation of *weights* for each particle is the likelihood calculations, but it is easy to calculate the likelihood of the sensor measurements given the output of its prediction step as the prior beliefs. If the calculated likelihood is less than a reasonable threshold (e.g., less than 1%), it is highly likely that the sensor produced a

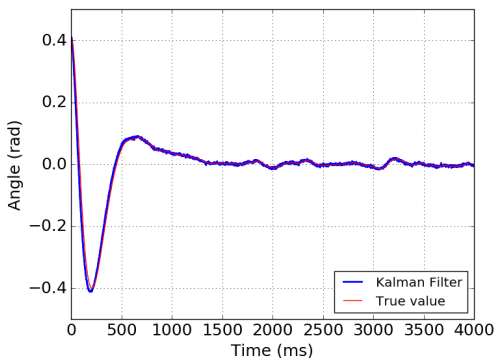


(a) Angle Tracking

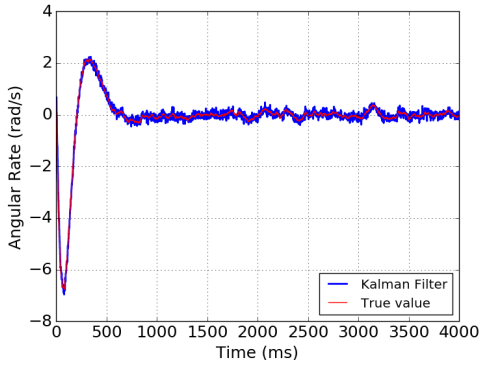


(b) Angular Rate Tracking

Figure 4.3. Kalman Filter for Sensor 1 ((a)-(b))



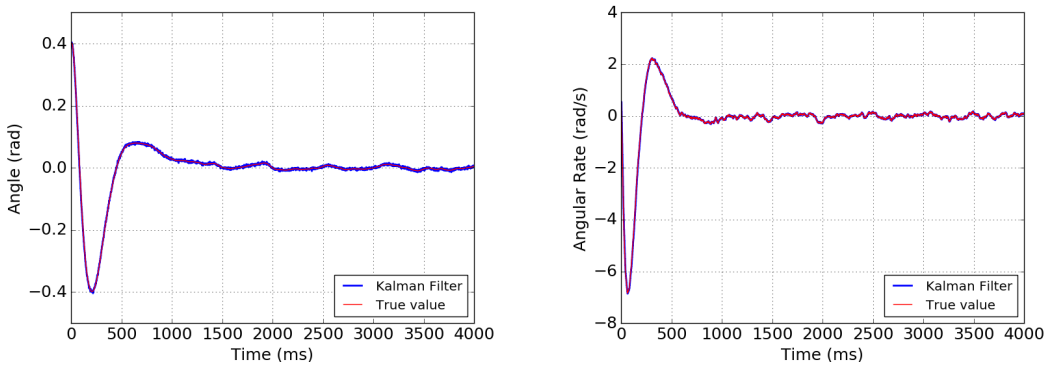
(a) Angle Tracking



(b) Angular Rate Tracking

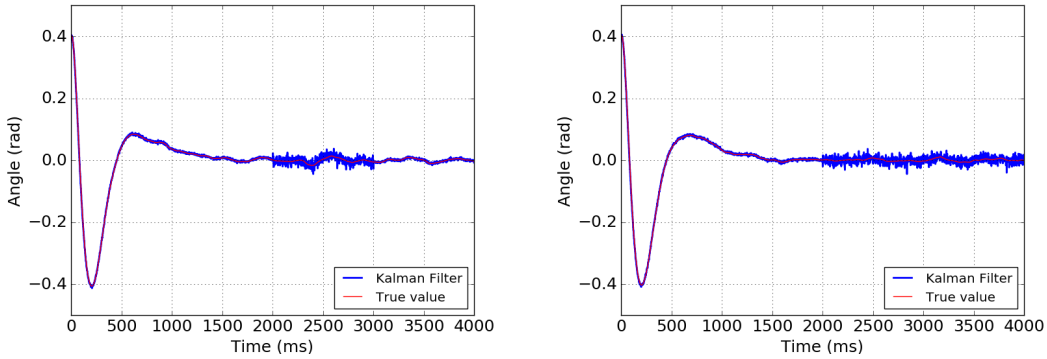
Figure 4.4. Kalman Filter for Sensor 2 ((a)-(b))

wrong value. In such a case, the system will skip the update step and take the value from the prediction step for this particular sensor. Intuitively, this approach assumes that the beliefs of the system have a certain amount of ”inertia” that would overcome the temporary sensor failures. With respect to a persistent sensor failure, this approach would consistently use the prior belief or the remaining working sensor(s) for the tracking. Figure 4.6 shows how the filtering algorithms would handle sensor failures. We deliberately assign some arbitrary wrong value (100) for sensor 2 for 1 second for the transient failure case, and for the remaining of the simulation for the persistent failure case. We see from the figures that



(a) Angle Tracking (b) Angular Rate Tracking

Figure 4.5. Kalman Filter for Two Sensors Combined ((a)-(b))



(a) Transient Failure (b) Persistent Failure

Figure 4.6. Kalman Filter with Sensor 2 Failure Model ((a)-(b))

the recovery from transient failures can be very fast, and performance degradation is not very severe.

4.2.6 Estimating Thermally-Induced Aging

TAAF expresses by how much the natural circuit aging process is accelerated by operating at a high temperature [24]. AdaFT uses a first-order thermal model to estimate temperatures; if desired, this model can be replaced by the user with one that more precisely captures the thermal characteristics and the failure dependencies of the particular hardware.

Each processor in AdaFT is treated as a single node, dissipating $p(t)$ Watts at time t. A standard equivalent electrical circuit model is used to model heat flow, where resistances and capacitances have thermal counterparts [49]. Thermal capacitance is the amount of heat required to raise the temperature of a node by one degree; thermal resistance determines the heat flow across a given temperature gradient (temperature is treated as an analogue of voltage). Denote by R and C the thermal resistance (associated with heat flow from the node to the ambient) and capacitance (of the node), respectively. Let $T_{proc}(t)$ and T_{amb} denote the absolute temperature of the processor and the ambient temperature, respectively. Then, the following differential equation emerges from the equivalent circuit model: $C\frac{dT_{proc}(t)}{dt} = p(t) - \frac{T_{proc}(t)-T_{amb}}{R}$. Solving this yields the temperature at any given time as a function of the power consumption.

The aging acceleration model for the hardware depends on the actual technology used. AdaFT provides the option to define a software module which expresses this function; however, a default aging module is provided based on the widely used Arrhenius acceleration model, in which the aging factor at time t, $\lambda(t)$, is proportional to $\exp(-E_a/(kT_{proc}(t)))$ [58]. Here, E_a is the *activation energy* [58], whose value is a user-provided input. The accumulated aging over a given interval $[a, b]$ is then calculated as $\int_a^b \lambda(t)dt$.

AdaFT computes TAAF based on the power consumed as a function of the load. The hardware configuration consists of three or more cores/processors on which tasks can be scheduled. The default scheduling policy is to pick the coolest processor to run at each time step; but the user can replace this scheduling algorithm by any other. AdaFT then computes the average TAAF as well as the instantaneous TAAF for each core. Recall that when the controlled plant is in sub-space S_i, it schedules i copies or versions of the control task.

It should be noted that TAAF is closely related to a more common term in the fault tolerance and reliability literature, i.e., the *mean time to failure* (MTTF), which is calculated as: $\int_0^\infty tf(t)dt$, where the $f(t)$ is the probability distribution density of the lifetime under unstressed conditions. Therefore, if the effective age of the device at chronological time t

is given by $x(t) = \int_0^t \lambda(\tau)d\tau$, the updated MTTF is given by: $\int_0^\infty tf(x(t))dt$. Once TAAF is calculated, AdaFT uses these equations to compute the MTTF.

Remark 4: Heating is by no means the only accelerator of failure. Other stressors include humidity, mechanical vibration and static discharge. Our focus in AdaFT is on allocating and scheduling computational workload which primarily affects device temperature. Other stressors have to be dealt with by other, orthogonal, means, such as improved packaging, mechanical damping and changes in circuitry; their impact on reliability can be modeled separately.

4.2.7 Sub-space Classification

During operation, the application must rapidly determine which sub-space it is in. This is a classification problem which must be solved rapidly.

Since such a real-time classification can never guarantee 100% accuracy, a conservative approach should be developed for system safety. The system might be allowed to make wrong decisions from S_1 to S_2 or to S_3, but not the other way around. Mis-classification from S_1 to a higher level of fault tolerance will do no harm to the system safety, only waste some resources.

Since the plant state-space is well defined, we can treat this as a supervised classification problem [38]. We first perform some pre-processing techniques such as feature scaling, feature selection or extraction using *principal component analysis* (PCA) [38]. We experimented with a variety of techniques including *random forest* (RF), *logistic regression* (LR), *neural network* (NN) and *support vector machine* (SVM) with various kernel functions, including linear, polynomial and Gaussian kernels [38]. Each algorithm has several hyper-parameters to tune, such as the number of trees for random forest, the regularization strength for LR, NN and SVM, etc. We use the technique of *grid search* to find the best possible algorithm with the best combination of hyper-parameters. Sometimes it is necessary to use an ensemble approach to find the best machine learning model, i.e., to use

several weaker individual models combined with *majority voting* for a stronger model. It should be noted that, although AdaFT has an interface for classification, all of the machine learning processes are application-specific and cannot be generalized. The interface is only for the purpose of connecting the fitted model to the analysis part of AdaFT for execution. We refer the reader to [38] for a detailed explanation of how these algorithms work. Here we concentrate on how to apply these algorithms to our specific needs.

We first discuss about the training data and features. These depend on the system. The feature list is the state vector plus the control period. For example, in modeling an anti-lock braking system, only the vehicle speed and the wheel speed of one quarter of the car (we used a quarter-car model for ABS control in a straight line) are required, therefore the features are these two state variables plus the ABS control period. As for the training data, it depends on the choice of SSC, the control period, and the physical dynamics of the system. For example, if the size of the state spaces that satisfies SSC is small, and the control algorithm period is long, it would result in small sub-spaces S_1, S_2, S_3. Normally, AdaFT will keep the default settings for the data size and the number of features. But if the training accuracy is very low under some predefined threshold, it will decrease the granularity of the simulation data. For example, in Figures 5.10 and 5.11, the granularity is 1 m/s for the speed. If the training accuracy is low, AdaFT will refine the granularity to a certain extent, such as 0.2 m/s. The learning algorithm will then re-run with the new granularity. This is similar to the approach in Section 4.2.4.

Next, we discuss about how to deal with Safety Critical Issues: We must ensure high *precision*, i.e., the proportion of the predicted positives that are actually positives. We can sacrifice some of the *recall*, that is, the proportion of true positives that are correctly predicted as such. With a slightly low recall, the system wastes some of the computing resource for unnecessary fault-tolerance redundancy; with low precision, potential hazards might occur.

One approach is to adjust the threshold value used to make decisions. Normally, the learning algorithm will produce a 1, for a two-class classification problem, if the output of the hypothesis function is larger than a threshold of 0.5, and a 0 otherwise. In multi-class classification problems, the algorithm will pick the class with the largest output from the hypothesis function. These probabilistic values show how confidently the algorithm makes certain decisions. If the confidence level of the algorithm needs to be increased, this threshold can be adjusted from 0.5 to a higher value. If there is any wrong classification from a more dangerous sub-space (e.g., S_3) to a safer sub-space (e.g., S_2 or S_1), the threshold value will be iteratively adjusted. If the largest value among all classes from the hypothesis function is higher than the threshold value, the algorithm will take that value and make the corresponding decisions; otherwise, it will determine the current system state to belong to S_3. Some machine learning libraries such as *scikit learn* provide automatic methods that can be used to find the best threshold value, such as the precision-recall curve [8].

Remark 5: AdaFT uses machine learning techniques to classify states into sub-spaces economically and efficiently. This well-known classification problem has long been the focus of researchers in the AI and machine learning community, and these are the most effective techniques to date. Which of these techniques will perform best depends on the application. If the true underlying decision boundary is complex and highly non-linear, probably a simple linear classifier such as logistic regression might not perform very well, and a more sophisticated model such as neural network or support vector machine with non-linear kernels should be used. With each specific application, one should use some search technique such as a grid search to find the best classifier/model with the best hyper-parameters associated with it. If high prediction accuracy cannot be guaranteed, then sufficient precision must at least be satisfied, by using some techniques such as the precision-recall curve to adjust the threshold with which to predict a particular class. The details are discussed in the case studies.

Remark 6: It should be noted that when applying to a running real-time system, AdaFT will need to predict the level of fault-tolerance, and possibly the version of each task during runtime, which would inevitably introduce overhead, even with machine learning techniques. One should be very careful to choose an appropriate classifier for the runtime prediction. This classifier should not only have high accuracy and/or precision, but it should also introduce only light overhead in terms of execution time and power consumption. Typically, given a particular micro-controller, the speed of the processor is fixed, unless *dynamic voltage and frequency scaling* (DVFS) is used. Therefore the complexity of the classifier in terms of the running time is particularly important. Normally, machine learning models with better capabilities, such as more hidden layers for a *neural network*, or number of trees for a *random forest*, will need more running time for accurate prediction. As a rule of thumb, *decision tree* based models will have very small running time, and therefore little overhead. In general, a model with less capacity would have small overhead [38].

4.2.8 Load Tuner

Sub-space generation is one of the core parts of AdaFT, its purpose is to determine the minimum number of copies for each task that is required in order to satisfy the safety conditions. This is one component of our *Load Tuner* design. Another component is to determine the version for each task, to balance the *quality of control* and the safety conditions, while at the same time to keep the power consumed by the computing resources as low as possible.

Typically, the safety constraint guaranteed by the sub-space determination makes sure that the physical plant can always operate in a safe condition. However, most of the time there also is a quality of control (QoC) constraint. For example, in a typical highway platoon, as long as the inter-vehicle space is greater than a certain threshold (e.g., 15 meters), it is safe, but if the space is larger than another value (e.g., 50 meters), the quality of control, in this case the highway throughput, is poor. Automatic highway control tasks such

as *adaptive cruise control* (ACC) algorithms control such inter-vehicle distance. More advanced ACC will yield better quality of control than simple ACC, although both could ensure the minimum safety inter-vehicle distance.

We now define the role of the load tuner in terms of task version selection and the number of task copies determination. Task version selection refers to deciding which version to choose for each control task. Number of task copy determination means deciding how many copies of each control task should be executed, using the parameters from the offline classification training for the sub-space generation.

The objective of the version selection is to decide which version of a control task to use, in order to satisfy the QoC constraints while keeping the TAAF as low as possible, assuming the periods of different versions of a task are the same, and assuming that a better version would have a worse TAAF. This can be viewed as a classification problem. The feature list here consists of the state vector, the versions to be used for each task, and the times at which these control tasks are applied to the plant. Normally this time feature is the period of a particular task for which we would like to decide to select a particular version.

Now suppose a system consists of a total of n tasks. At a particular time instance, there are m tasks with new iterations becoming available. We would like to decide which version to run for each of the m tasks. Unfortunately this problem is NP-hard and intractable, since the complexity of finding a global optimal solution for the m task combination is $O(v^m)$, where v is the number of versions. We provide a heuristic here to find a sub-optimal solution for the version selection, with the complexity of $O(vm)$. Algorithm 3 is used for this purpose.

We initialize the task versions as the ones with the worst QoC. For each task, we incrementally check a better version, using the trained classifier with all other task versions fixed. If, during the process, the QoC is satisfied, we are done with this task and move to the next one. This algorithm might sometimes be stuck at local optima.

ALGORITHM 3: Load Tuning

Input: The state vector and control tasks
Output: Versions and copies for each available task;
Initialize the versions of all available m tasks as the one with worst QoC, and keep the versions of the remaining $n - m$ tasks as is;
for *Each of the available* m *tasks* **do**
  for *Each version ordered from worst to best QoC* **do**
    if *the current version is the best version* **then**
      Use the sub-space classifier to determine the number of copies;
    end
    Use the system model to predict the QoC after one period of this task, using the current version assignment;
    if *the QoC constraint is satisfied* **then**
      Use the sub-space classifier to determine the number of copies;
      Break;
    end
  end
end
return the current version and copy assignment;

Number of Copies Determination means picking the number of copies for each task during runtime. Notice that this part can be decoupled from the Version Selection part, since the worst case control and the zero/default control inputs are the same, regardless of the versions of the tasks. Therefore Algorithm 3 also deals with the copy assignment for each task.

Note that in the case of a multiple-version multiple-control system, the complex version would normally have a larger S^3, due to its better QoC. Therefore, each of the m versions for every n control tasks would have a different S^3 and sub-spaces. To generate all these sub-spaces even offline calculation is intractable once n and m become large. If we include the version number for each task in our machine learning model, the feature dimension would grow exponentially. Thus, for computational purposes, we will use the simplest version for each task to generate S^3, assuming that if the simplest version for each task can guarantee a safe control, the more complex version with a better QoC would also guarantee a safe control. This S^3, although conservative, can guarantee a safe control as long as all

of the control tasks are correct (does not matter which version). After S^3 is generated, we can use the approach introduced in Section 4.2.3 to generate all the sub-spaces.

4.2.9 Real-Time Computing Model

AdaFT has an built-in real-time task model which has the following attributes: name, period, deadline, *worst case execution time* (WCET), *actual execution time*, power, and status (running, idle, etc.). It also has a probability density function of the execution times, in order to randomly generate the actual execution time for simulation purposes. The period and deadline of the task are needed for real-time scheduling, and the power is needed for thermal aging (TAAF) analysis.

Real-time scheduling is an essential part of real-time system control. After the Load Tuner has determined the version and number of copies for each task, the scheduling algorithm will assign the tasks to appropriate cores to run. There are two widely used scheduling algorithms, i.e., the *rate monotonic* (RM) and the *earliest deadline first* (EDF) algorithm. The main scheduling algorithm used in industry today is RM, which determines the task priorities as inversely proportional to task periods [15][28]. EDF determines the task priorities according to their absolute deadlines.

AdaFT has scheduling modules to support both RM and EDF. For a system composed of multiple cores/processors, AdaFT uses the widely developed EDF-based algorithms: *Global EDF* (GEDF) and *Partitioned EDF* [15]. GEDF assigns m out of n highest priority tasks to m available processors during runtime. GEDF has also the property of automatically balancing the workload for each CPU. On the other hand, Partitioned EDF requires the designer to assign tasks to particular processors offline according to EDF. During the execution, when a new iteration of a task arrives, the scheduler just assigns this task to its corresponding processor.

When users develop their CPSs using AdaFT, they must guarantee the system schedulability (i.e., that all task deadlines will be met). RM, EDF and GEDF have been well

studied for schedulability, interested readers can refer to [15] for the details, as well as a comprehensive survey of other real-time scheduling algorithms.

With this real-time scheduling model, the user can easily experiment with different periods and possibly different execution times of each control task, and see the impact of these parameters on the size of each sub-space.

Remark 7: Regarding communication faults, highly effective coding mechanisms exist to reduce them. The level of coding can be adjusted according to the reliability requirements of the task in question. Another alternative is to send multiple copies. The AdaFT framework can be used to determine the impact of such choices (and others) on the rate of degradation of the hardware. If multiple copies of individual message need to be sent, and if this imposes non-negligible stress on active (powered) elements, the message sending and receiving operation can be treated within AdaFT as tasks with the core generating the message. No change to the framework is therefore required.

For example, consider the stochastic communication paradigm introduced in [7]. The idea is to send a message to a random intermediate destination from where it is forwarded to its destination. This requires the creation of additional send-and-receive tasks which can be handled by AdaFT as any other task.

Note that AdaFT is equally applicable to all forms of redundant hardware, software, time and information. The framework determines from the application dynamics whether the output needs to be fault-masked, fault-detected (without masking) or neither. What redundancy types are used (individually or in combination, e.g., N-version programming with triplex) is up to the user. Based on the FT level, the framework calculates the stress placed on the computational elements.

Remark 8: AdaFT in this dissertation is mainly for simulation purposes. In order to apply AdaFT to real-world applications, some practical issues need to be addressed. The first one is that AdaFT needs a fairly accurate physical dynamics model of the plant. Computer-controlled systems of the physical plant already require detailed and accurate dynamic

models for implementation. Many design decisions are based on such models. For example, designers must determine the appropriate task periods for the periodic control tasks. Doing so requires a tradeoff between the quality of control and the imposed computational workload. Another decision which requires accurate modeling, is the minimum time between the releases of sporadic tasks in a real-time system. Evaluation of the comparative effectiveness of various competing control algorithms also requires accurate models. Such models are also needed in the certification of life-critical cyber-physical systems.

The control engineering community has responded to such a need by constructing detailed models of controlled plants. For example, CarSim [48] is based on a validated, highly detailed model of key aspects of automotive dynamics, used by both industries as well as universities. Similarly, detailed models of aircraft exist, such as [53].

There are many tools for system dynamics modeling. One example is to use regression techniques to model the input-output relationships given enough data collected by running the actual hardware and physical plant. Matlabs identification toolbox shows numerous examples of how to derive, both linear and non-linear, dynamic models from data.

The second issue is where to incorporate the AdaFT's *load tuner*, which includes both version selection and number of copies determination. The easiest and most straightforward way is to incorporate AdaFT into the scheduling model of the RTOS. Before the actual scheduling, AdaFT will determine the task load by predicting which version of the new task, as well as how many copies this task will need for the current physical state. Then, the scheduling algorithm will schedule the determined task load for the system. As long as AdaFT's load tuner runtime overhead can be kept reasonably small, it will not affect system performance.

4.2.10 DVFS

Energy efficiency in CPSs is achieved by adjusting the power consumption of the system. Recently emerged real-time dynamic voltage and frequency scaling (RTDVFS) tech-

nique is one of the most promising techniques for energy-efficient scheduling. Real-time applications often have large variations in terms of their actual execution times. As a result, they often finish much earlier than their estimated worst-case execution time [2]. RTDVFS-based techniques exploit these variations for dynamically adjusting voltage and frequency in order to reduce power consumption of processors. The challenge for these techniques, however, is to preserve the feasibility of schedule and provide deadline guarantees.

In [5], the authors proposed several RTDVFS that are suitable for multi core/processor real-time systems. AdaFT implements one of the algorithms discussed in [5], called *Dynamic Slack Reclamation* (DSR). The DSR algorithm is shown in Algorithm 4. It is based on detecting the early termination of tasks with respect to their worst-case execution times. Since it is not possible to know a priori the actual workload of a running task until it terminates, DSR algorithm determines the amount of slack only once a task terminates by comparing worst-case and actual-case workload. Based on the amount of available slack, if any, the algorithm determines how much a processor could be slowed down to execute the next dispatched task such that the dispatched task does not execute beyond its worst-case boundary defined in the canonical schedule. Algorithm 4 demonstrates that, for a feasible task set, S^{can} is obtained a priori and all tasks are sorted according to their priority order under the selected scheduling algorithm. The highest priority m tasks are assigned to m processors for execution at statically specified maximum frequency (f_{max}) and dynamic slack is initialized to zero. Note that DSR updates current frequency (f') only at scheduling events. If a task terminates, its earliness is determined by computing dynamic slack time based on the difference between actual and worst-case (canonical) execution times. For positive slack (slack greater than 0), the total amount of available time and required time for next priority task at f' of the processor is determined. Based on the difference between available and required time, a scaling ratio (SR) is computed to update f' for subsequent task. However, if the terminating task does not generate dynamic slack, statically determined f_{max} is retained for next priority task.

ALGORITHM 4: Dynamic Slack Reclamation

Input: Control tasks profiles and priorities S^{can}
Output: Frequency of the processor, f';
Sort ready tasks in priority order;
Set $slack = 0$;
$f' = f_{max}$;
for *Each scheduling event* **do**
  if *scheduling event is termination* **then**
    Detect early termination;
    Compute dynamic slack (slack = C_i - AET_i);
    if $slack > 0$ **then**
      Compute available time ($t_{available}$)for next task at f';
      Compute required time (C_{i+1}) for next task at f';
      Compute scaling ratio ($SR = t_{available}/C_{i+1}$);
      Update f' w.r.t SR;
      if $f' < f_{min}$ **then**
        $f' = f_{min}$;
      end
      Execute next priority task at update f';
    end
    else
      $f' = f_{max}$;
    end
  end
end

4.2.11 Hardware Provisioning

Controllers of life-critical plants must be designed in such as way that the overall reliability is higher than a user-defined requirement, while making sure that the quality of control (QoC) satisfies the constraints, and the amount of hardware provisioned is as small as possible.

There are two types of systems, repairable and non-repairable. For repairable systems, the primary aim is to keep the QoC as high as possible so that it can satisfy the QoC constraints, under the assumption that if one or more cores do fail, the system can soon be repaired. In such a system, the simpler version of the control task is only used when the overall system reliability is lower than a threshold, so that the system can operate in a safe degraded mode.

Non-repairable systems (e.g., in space missions), always have a limited operational horizon or mission lifetime, which can be expressed in terms of the *mean time to failure* (MTTF). The design objective for such systems is to resolve the trade-off between the (a) QoC provided, (b) mission lifetime or MTTF, and (c) amount of hardware provisioned.

AdaFT can be used to design non-repairable CPS. As mentioned in Section 4.2.6, AdaFT will compute the MTTF after TAAF is calculated. AdaFT will take the QoC constraint as one of its inputs,and will decide which version of each task to choose and how many copies to run, to ensure that constraint is satisfied. The user can follow an iterative approach to determine the amount of hardware (the number of processors in this case) provisioned as follows. The user starts with the minimum number of processors required, which is 3 since we need at least 3 processors in case of full level of fault-tolerance. Then with several initial conditions, possibly with random fault injections, the user can check if the MTTF after the simulation meets the requirements. If not, then one can incrementally adjust the number of processors, until the MTTF is satisfied.

4.3 AdaFT Programming Interface

Figure 4.7 shows the major parts of AdaFT, in a UML class diagram. AdaFT provides API function calls such as the sub-space generator in both Python and Matlab.The *CPS* class wraps all major components of a CPS, including the *Physical System*, *Sensor*, *Cyber System*, and *Actuator*. The *Sub-Space Generator* class uses the *CPS* to generate all of the sub-spaces that are then used by the *Classifier* to fit a classification model, which is later used by *CPS* for execution. Initially, *CPS* runs with correct, zero and wrong controls to generate sub-spaces, and then uses machine learning techniques to generate a fitted model. Once the fitted model (or classifier in our case) as the *classifier* is generated, it runs with the classifier for reliability analysis. The *Cyber System* includes one or more *Processor* objects, which has the support from *RTOS*, which in turn consists of several *Task Model* objects as

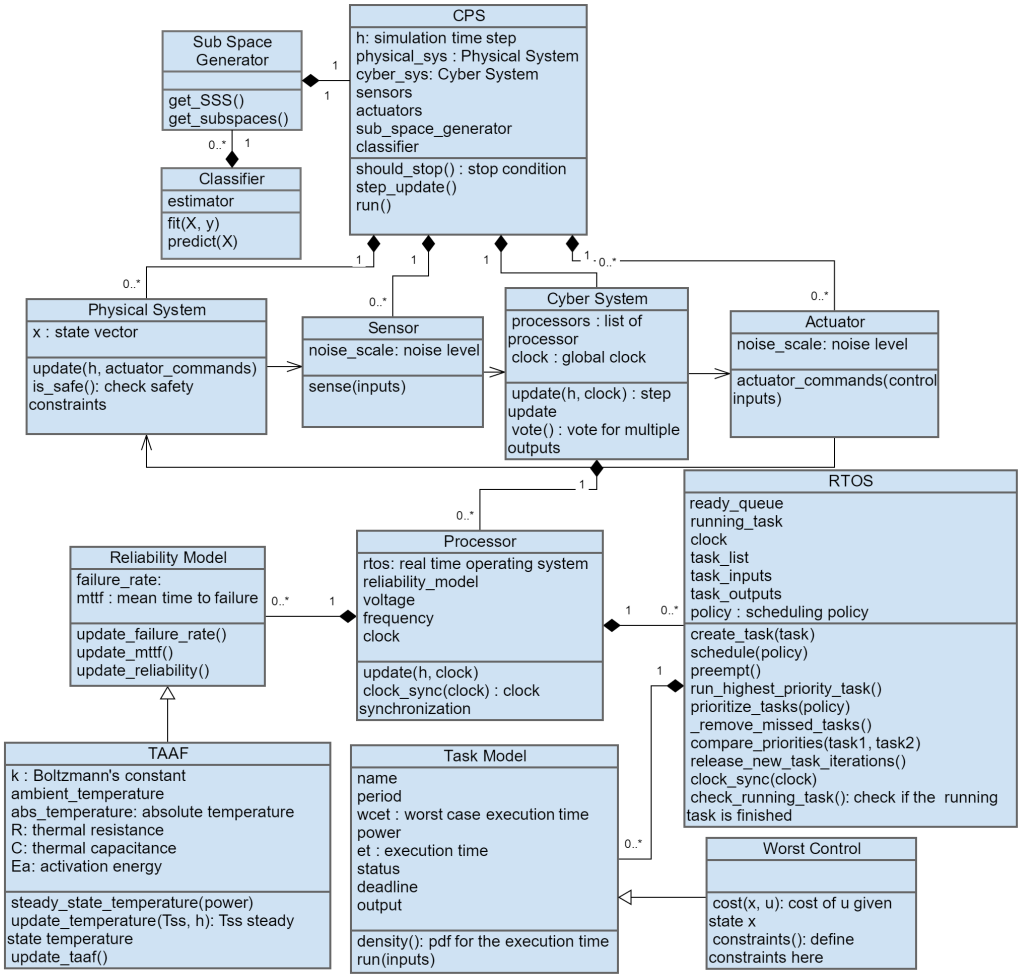


Figure 4.7. Block Diagram of AdaFT

real-time tasks. The *Worst Control* is a child-class of Task Model, and the *Reliability Model* and its child-class *TAAF* are also part of the *Processor* class.

To use AdaFT, one has to write a physical system implementation, possibly inherited from the *PhysicalSystem* class. In particular, the *update()* and *issafe()* methods need to be implemented. *update()* simulates the physical states updates, according to control inputs and the corresponding actuator signals. *issafe()* checks if the SSC is met during simulation.

The next step is to implement the control tasks, both correct and worst case control, through the API from *TaskModel* class. Essential attributes of a task need to be specified: namely, power, WCET, deadline, and period. In addition, the *run()* abstract method must

be implemented, which is the actual algorithm of the task. Note that the filtering algorithms discussed before also are real-time tasks and should always be run with the highest redundancy, since the physical side information will be estimated through them. The inputs of the custom control tasks should be the outputs of these filtering tasks, whose inputs are the raw sensor readings.

There are additional methods that users can implement or override the default implementation, such as the heuristics to sort the data points for the sub-space generation, according to some safety rules, but they are not required, either because they are not a core part of AdaFT, or AdaFT already has the default implementation.

If the system dimension is small, AdaFT will start the whole process to generate the sub-spaces through *getSSS()* and *getSubspaces()* methods. Otherwise, the user has the option to provide a fitted machine learning model discussed in Section 4.2.4 as the input to the *getSSS()* method to generate S^3.

After the sub-spaces S_1, S_2 and S_3 are generated, the user must select a machine learning algorithm for the classifications, through the API from the *Classifier* class. This fitted machine learning algorithm will then be used by the analysis part of AdaFT for reliability analysis. Other additional custom parameters the user can specify include, sensor and actuator noises, processor voltage and frequency, real-time scheduling policy for RTOS, etc.

4.3.1 Example Program

The inverted pendulum example is a linear system with a Kalman filter, and *A, B, C, D* matrices to define and track the physical states. Detailed equations can be found in the case study. Here we demonstrate how to program such a adaptive fault-tolerance system using AdaFT.

To provide the system dynamics, and the control task (LQR), we use two child classes inherited from their corresponding parent classes. Note that for the worst case control, if

the worst case output can be determined without solving an optimization problem, the user can override the *run()* method to directly provide this output, since the *WorstControl* class itself is a child class of *TaskModel*.

```
class Pendulum(adaft.PhysicalSystem):
  def update(self, h, clock, u):
    self.x = # update the state vector according to actuator input u.

  def is_safe(self):
    return -0.5 <= self.x[2] <= 0.5
        # outside this range is unsafe

class LQRInvPen(adaft.TaskModel):
  def density(self):
    return max(0, min(self.wcet,
            np.random.normal(self.wcet / 2, 0.001)))

  def run(self, inputs):
    # inputs are the outputs from kalman filter
    # this is the control input from LQR algorithm
    self.output = -np.dot(self.K, inputs)

class WorstLQRInvPen(adaft.WorstControl):
  def cost(self, x, u):
    new_x = # get new state values with control u and state x
        return 1/2 * ((new_x[2] - desired_x) ** 2)

  def constraints(self):
    self.constraints = {# define constraints here}
```

The user can specify sensor and actuator noises, then insert these objects into the *CPS* driver class, to start generating sub-spaces, and other tasks.

```
angle_noise = # specify sensor noise for angle measurements
rate_noise = # specify sensor noise for angular rate measurements
actuator_noise = # actuator noise

sensor = adaft.Sensor([angle_noise, rate_noise])
actuator = adaft.Actuator(actuator_noise)

lqr = LQRInvPen()
worst_lqr = WorstLQRInvPen()
task_list = {lqr.name:lqr}
rtos = adaft.RTOS(task_list)
# We create 3 processors for redundancy
processors = [adaft.Processor(rtos) for i in range(3)]
cyber = adaft.CyberSystem(processors)
```

```
cps = adaft.CPS(sensor, actuator, pendulum, cyber)
subspace = adaft.SubSpaceGenerator(cps, {lqr.name:worst_lqr})
subspace.get_SSS()
subspace.get_subspaces()

clf = adaft.Classifier()
# perform machine learning techniques here
clf.fit(subspace.subspaces)

cps.classifer = clf

cps.stop_condition = # determine when the simulation should stop
cps.pendulum.x = # define initial state conditions here
cps.run()

# now we can see TAAF and/or MTTF of each processor
plot(cps.cyber.processors[0].taaf)
print(cps.cyber.processors[0].mttf)
```

CHAPTER 5

CASE STUDIES

In this section we present four case studies using AdaFT. The first one is a linear system, the second and third ones are non-linear systems, with the third one being a multi-agent system, and the last one is an linear multiple-control multiple-version system. We use a data analytic approach to analyze case 1; and show the impact of different environmental conditions on sub-spaces. For case 2, we show how interactions among multiple agents can be studied through AdaFT. Finally, in Case 4 we show an end-to-end analysis using AdaFT on a humanoid robot system, with multiple control tasks.

5.1 Case 1: Computer Controlled Inverted Pendulum

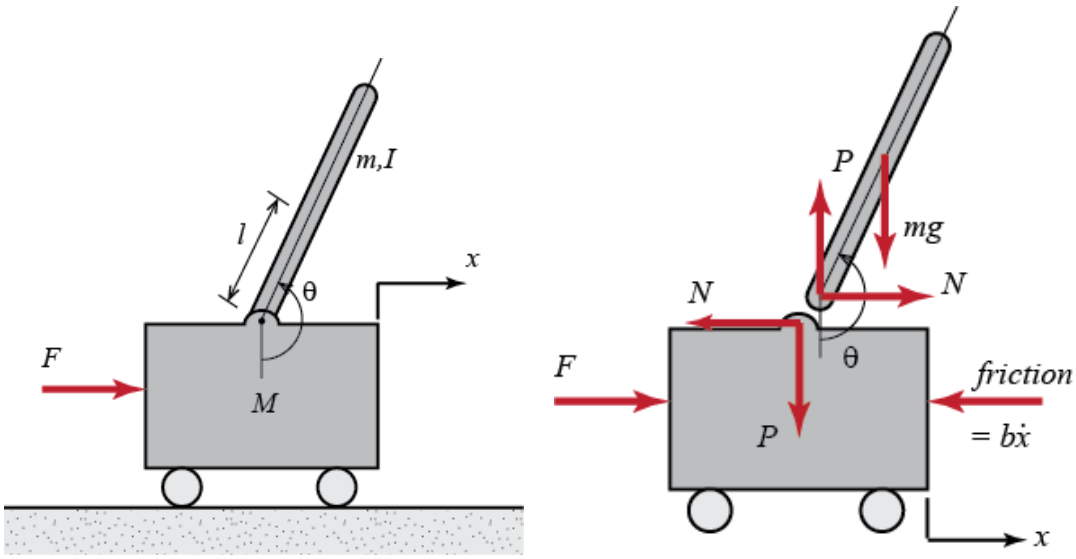
The system in this example consists of an inverted pendulum mounted on a motorized cart. The inverted pendulum system can be modeled using either the *linear time invariant* (LTI) or the nonlinear form. The LTI form uses the following set of linear equations.

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{5.1}$$

$$y(t) = Cx(t) + Du(t) \tag{5.2}$$

where $x(t), y(t)$ and $u(t)$ are the plant state, output vector, and input vector, respectively. A, B, C, and D are the matrices defining the controlled plant.

The objective of the control system is to balance the inverted pendulum by applying a force on the cart. A real-world example of this inverted pendulum system is the attitude



(a) Inverted Pendulum (b) Force Analysis

Figure 5.1. Inverted Pendulum Cyber-Physical System [35]

control of a booster rocket at takeoff. In particular, this case study uses an optimal control algorithm *linear quadratic regulator* (LQR) [59] to control the cart's position. In this case study, we make the following assumptions.

1. Actuator noise for both cart position and pendulum angle is 3 N (Newton) as the standard deviation of the measurement (with the mean value as the true value). This means that the actuator will randomly output a control force with the underline true value as the mean, and 3 N as the standard deviation.

2. Sensor 1 noise for the four states (cart position (m), velocity (m/s), pendulum angle (rad), angular rate (rad/s)) are: (0.001, 0.001, 0.01, 0.005) as the standard deviation (with the mean value as the true value).

3. Sensor 2 noise for the four states are: (0.001, 0.001, 0.002, 0.1) as the standard deviation.

4. Initial condition for TAAF analysis is (cart position (m), velocity (m/s), pendulum angle (rad), angular rate (rad/s)): (0, 0, 0.4, 0.5).

5. Kalman filter is used for filtering out sensor noises.

6. LQR control task period: 20 ms; deadline 20 ms; WCET: 2 ms

The pendulum is assumed to move in a two dimensional vertical plane, as shown in Fig 5.1. For this system, the control input is the force F that moves the cart horizontally. The outputs of the control are the angular position of the pendulum θ and the horizontal position of the cart x.

We further assume the following quantities:

1. Mass of the cart is M = 0.5 kg.

2. Mass of the pendulum is m = 0.2 kg.

3. Coefficient of friction for cart is b = 0.1 N/m/sec.

4. Length to pendulum center of mass is l = 0.3 m

5. Force applied to the cart: F

6. Cart position coordinate: x

7. Pendulum angle from vertical (down): θ

5.1.1 Simulation Setup

Figure 5.1(b) is the free-body diagram of the two elements of the inverted pendulum system.

We refer the reader to [35] for the detailed mathematical state space equations for the system. We use a standard form of the state-space matrix to represent the physical dynamics, as shown below, which includes a series of first order differential equations.

In this system, there are two outputs: the cart's position and the pendulum's position. Therefore, the C matrix has two rows. The cart's position is the first element of the output

$\mathbf{y}$ and the pendulum's deviation from its vertical position (a.k.a 0 degree position) is the second element of $\mathbf{y}$.

Below is the summary of the linear equations with the values.

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\phi} \\ \ddot{\phi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -0.1818 & 2.6727 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -0.4545 & 31.1818 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \phi \\ \dot{\phi} \end{bmatrix} + \begin{bmatrix} 0 \\ 1.8182 \\ 0 \\ 4.5455 \end{bmatrix} \vec{u} \tag{5.3}$$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \phi \\ \dot{\phi} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \vec{u} \tag{5.4}$$

We use the *lqr* function from Matlab to get the K matrix, which implements the LQR control [59]. This function computes the control gain matrix K, with the inputs of system dynamics matrices, the Q and R matrices. The Q matrix defines the cost experienced when the states deviate from the desired ones, whereas the R matrix defines the cost when the control inputs are large [59].

We define the SSC to be:$-0.5 < \phi < 0.5$, where ϕ is the angle of the pendulum, as we don't want the pendulum's angle to be too large, otherwise it is unsafe.

As for the cost function for the wrong controller or actuator output, we first define the desired states for the system. The aim of the inverted pendulum system is to keep the pendulum as close to vertical as possible, therefore, the desired state of the pendulum angle is 0. Then, we select the cost function as follows: $cost = (\frac{\phi - \phi_d}{\phi_d})^2$. ϕ_d is the desired pendulum angle. This inverted pendulum example has 4 state variables, but we only care about the third state which is the pendulum angle.

We implemented this system using the AdaFT API, as shown in the example program template from Section 4.3.1. We now discuss some of the results.

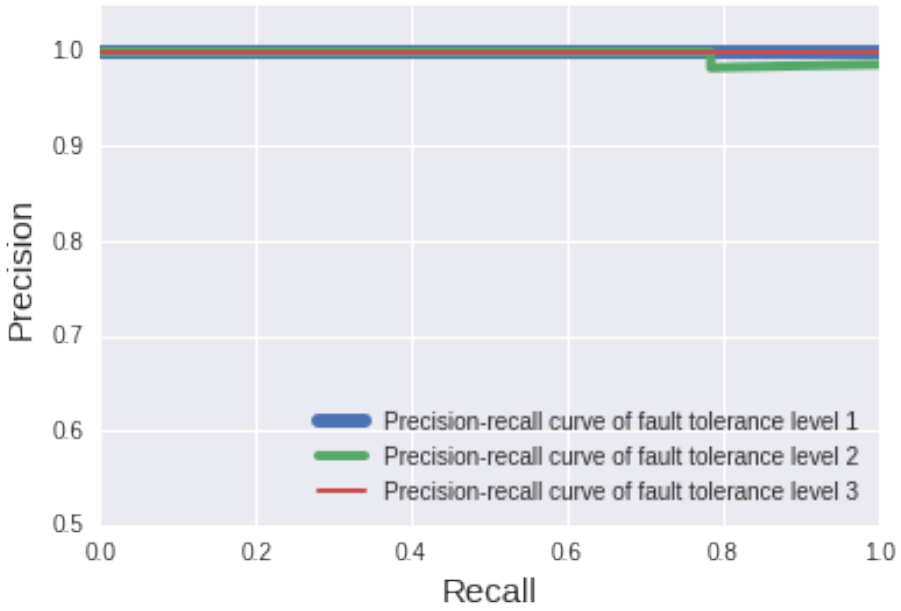


Figure 5.2. Precision-Recall Curve for the Classification of the Inverted Pendulum System. Here we use the Random Forest algorithm as an example. In fact, most of the fitted algorithm for this system have similar prediction performances.

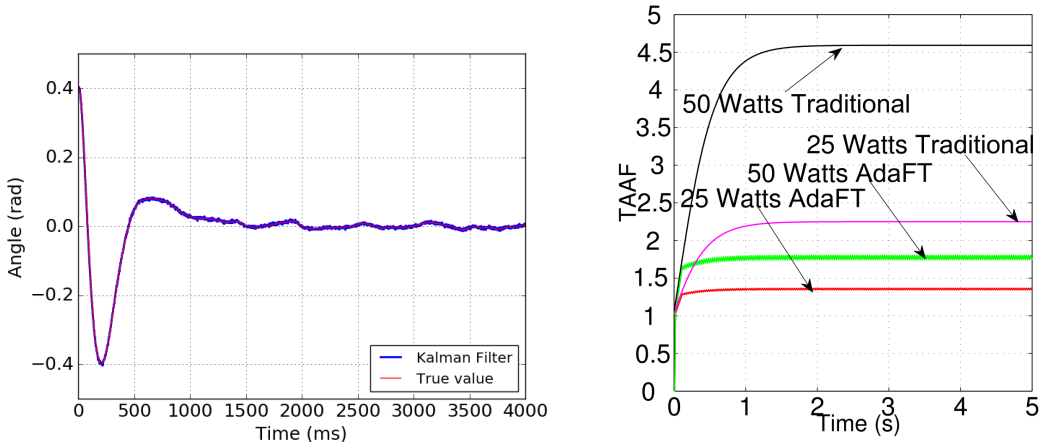
5.1.2 Results

We followed the approach presented in Section 4.2.4 to generate S^3, which was then used to generate all three sub-spaces.

We have collected data for the following operating space of the system:

1. Cart position: from -6 to 6 meters
2. Cart velocity: from -6 to 6 m/s
3. Pendulum angle: from -0.5 to 0.5 radians, outside which we considered as unsafe.
4. Pendulum angular rate: from -1 to 1 rad/s

After obtaining the sub-spaces, we experimented with two cases regarding S_3. The first one only considers S_3 inside S^3, while the second one allows S_3 to include data points inside S^3 that don't belong to either S_1 or S_2, as well as all data points outside of S^3. If, for



(a) Pendulum Angle (rad) (b) TAAF

Figure 5.3. Inverted Pendulum TAAF ((a)-(b)). (a) shows the pendulum angle trajectory during the simulation with this particular initial condition, as well as the Kalman Filter inference results, while (b) shows the TAAFs under different configurations.

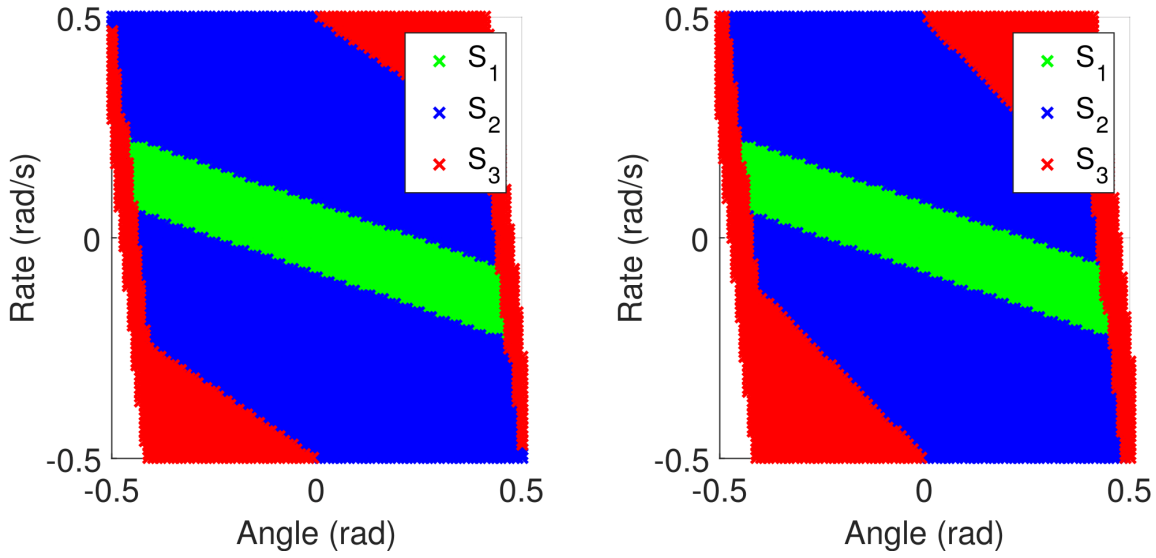
Table 5.1. Random Forest Performance for Inverted Pendulum

	Feature Importance				Accuracy		Memory	Time
	Position	Velocity	Angle	Rate	Training	Testing		
Entire Space	0.24	0.3	0.21	0.25	100%	99.7%	6.2kB	1ms
Within S^3	0.0038	0.01	0.97	0.012	100%	100%	2.5kB	1ms

any reason, the states fall out of S^3, full level of fault-tolerance is required, which would rarely happen in practice as long as the control system is properly designed.

The results show that S^3 includes cart positions with the range from -1 to 1 meter, cart velocities from -4 to 4 m/s, pendulum angle from -0.5 to 0.5 radians, and angular rate from -1 to 1 rad/s. Inside S^3, S_1 has pendulum angles from -0.35 to 0.35 radians. Fig 5.5 (a) shows the cross section sub-space with both cart position and cart velocity fixed at 0 for simple visualizations.

As discussed in Section 4.2.7, we use several popular machine learning classification techniques to find the fitted algorithm with the best performance. We will focus on the accuracy here, since the storage and prediction time don't have large differences between machine learning algorithms. From our results, except simple logistic regression that only



(a) 20ms (b) 30ms

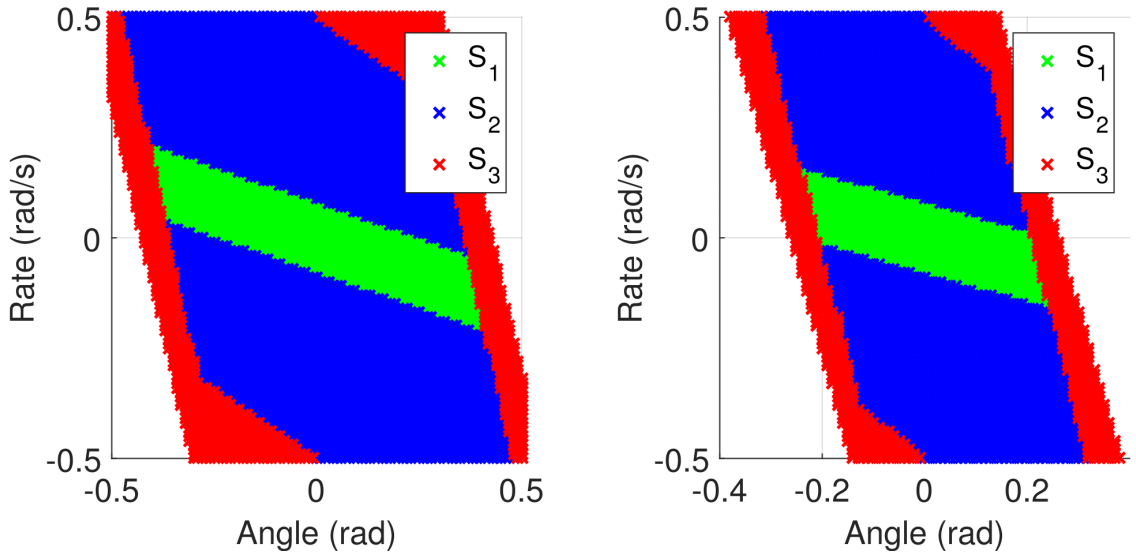
Figure 5.4. MPC Sub-space with the cart position and cart velocity both fixed as 0, for the purpose of visualization. It uses MPC with two periods of 20 ms and 30 ms. Fig 5.6(b) shows the difference of these two sub-spaces for better visualization.

Table 5.2. Decision Tree Performance for Inverted Pendulum

	Feature Importance				Accuracy		Memory	Time
	Position	Velocity	Angle	Rate	Training	Testing		
Entire Space	0.36	0.05	0.51	0.06	100%	99.6%	1.2kB	0.0052ms
Within S^3	0.0038	0.001	0.99	0.002	100%	100%	0.5kB	0.0052ms

has 85% of testing accuracy, most of other algorithms have high score close to 100%. We only show here the performance of random forest, since it also allows us to see the feature importance. From Table 5.1, the fitted algorithm using random forest has high testing accuracy. What is different from our initial guess is that the cart velocity and angular rate also play important roles for the case of entire space, other than the angle itself.

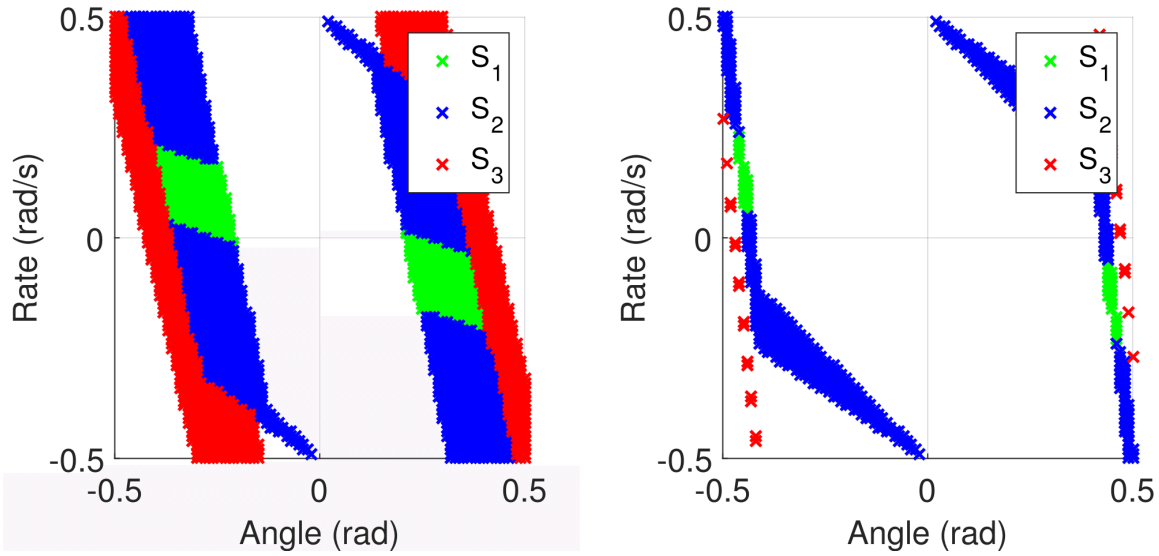
Table 5.1 and Table 5.2 show the memory usage when storing the fitted random forest and decision tree algorithm. We use Python's pickle module to estimate the entire fitted algorithm space. We can see here the decision tree algorithm uses much less memory,



(a) 20ms (b) 30ms

Figure 5.5. LQR sub-spaces with the cart position and cart velocity both fixed as 0, for the purpose of visualization for two periods of 20ms and 30ms.

with the same high prediction accuracy. With this amount of data stored in memory, which nowadays typically has hundreds of MBs, and with the very small running time overhead (e.g., for the decision tree, it is less than 0.01ms), this is a compact way to make accurate runtime predictions. In fact, this memory usage estimation is very conservative, since we are using the pickle module to estimate the entire fitted algorithm object in Python, which includes other unnecessary attributes such as the feature importance, references to other objects, etc. In a real design, we only need to store the rules in the memory. In this figure, we limited the max depth of the decision tree to 9, for better visibility, and this limited tree algorithm would have a testing accuracy of about 88%. In our real decision tree algorithm , we have a max depth of 15 and testing accuracy of nearly 100%. As we see in this figure, we only need to store this rule in memory, and the worst case execution time for a prediction is in linear relation with the max depth, which would typically be less than 50. Therefore, decision tree based machine learning algorithms are well suited for such real-time problems. Other machine learning algorithms would have similar performance.



(a) LQR (b) MPC

Figure 5.6. Sub-spaces Comparison for Different Periods but the Same Algorithm. (a) shows the sub-space regions that are in 20ms periods but not in 30ms periods for LQR, while (b) shows the similar information when the algorithm used is MPC.

For example, *Logistic Regression*, *Neural Networks* and *Support Vector Machine* would only need to store the trained weights into the memory, which would typically consist of a few hundreds of these weights (coefficients) depending on how complex the algorithm is. The prediction time for most algorithms, except *Deep Neural Networks* with many hidden layers and hidden units, is short.

Figure 5.2 shows the precision-recall curve for S^3. As expected we have a very high precision, except when the recall for S_2 is higher than 80%, its precision falls to 95%. AdaFT can adjust the prediction threshold to improve the precision. Once we set the precision threshold to 99.5% for S_2, we have a total testing accuracy of 98.61%; these are the errors that are mis-classified from S_2 to S_3. The precision-recall curve for the entire space is similar and is omitted here.

Regarding the reliability results, we experimented with two different power usages for the control task: 25 and 50 watts. Figure 5.3(b) clearly show that using AdaFT, there

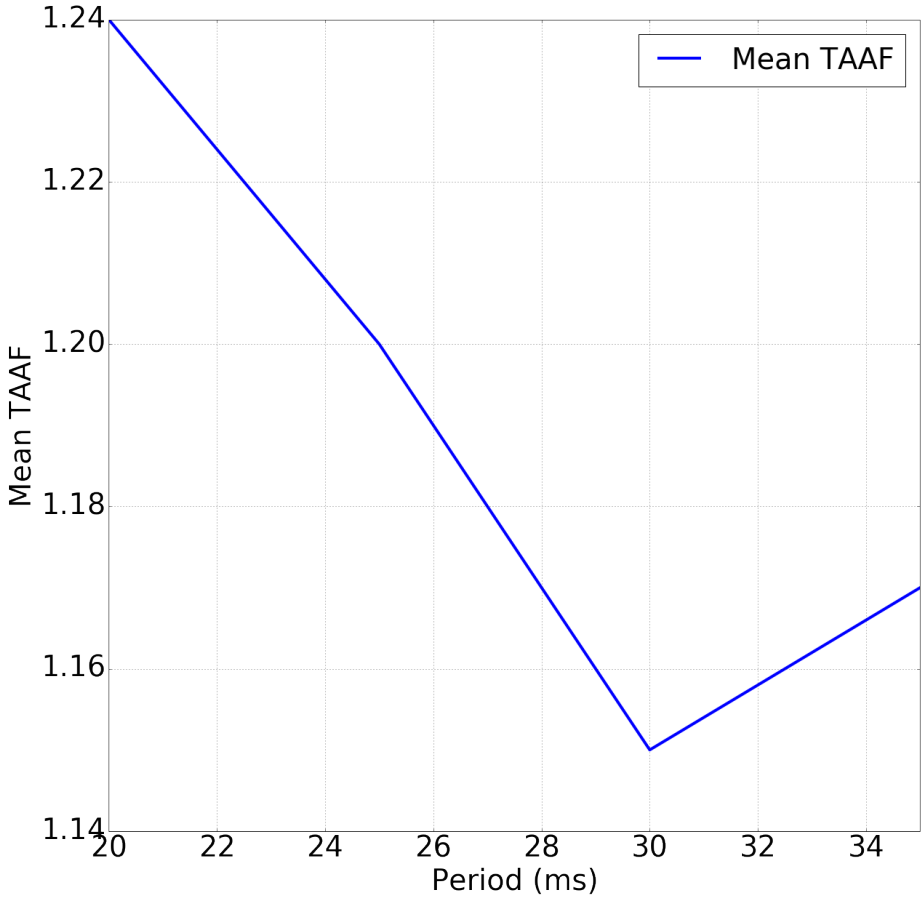
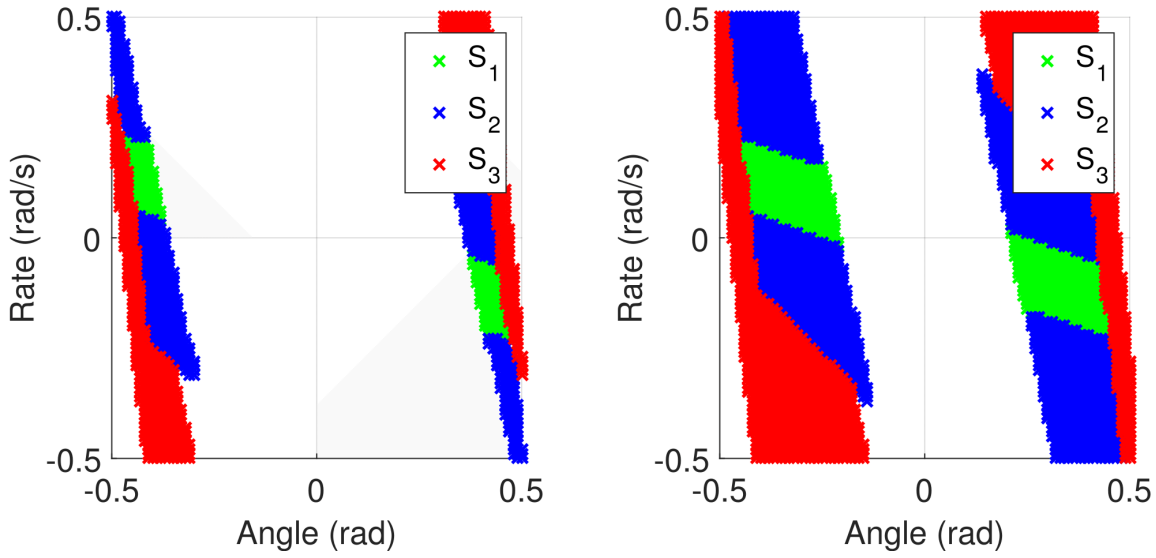


Figure 5.7. Mean TAAF with Different Periods of LQR Control.

is a significant improvement in TAAF compared to traditional FT. Under our scheduling algorithm (picking the coolest processor to run the task), all three processors have similar values of TAAFs. Note that in reality 50 watts is a very high power consumption and it rarely exists for real-time embedded tasks. we still include here only for illustrating, theoretically, that the bigger the power consumption is, the more benefits there would be from AdaFT.

For the impact of QoC on Sub-spaces, as shown in Fig 3.1, different qualities of control will impact S^3. In fact, QoC will potentially have a significant impact on all the sub-spaces. Typically, a larger control period will yield a worse quality of control, and as a result, we can expect the impact of incorrect control inputs to be greater. Therefore, we will need



(a) 20ms (b) 30ms

Figure 5.8. Sub-spaces Comparison with Same Period but Different Algorithms. (a) shows the sub-space regions that are in MPC but not in LQR for 20ms period, while (b) shows the similar information except that the 30ms period is used.

to use a higher level of fault-tolerance for a greater fraction of the state space. Fig 5.5 and Fig 5.6(a) illustrate this by showing the size difference in sub-spaces using the LQR algorithm. In particular, Fig 5.6(a) shows by how much the sub-spaces are reduced when using a 30 ms period, compared to a 20 ms period.

However, if there is another version of the control task that uses more complex and better algorithm, the impact of a larger control period could be reduced. In other words, the size difference of the sub-spaces between periods will be reduced if the quality of the control algorithm improves. Fig 5.4 and Fig 5.6(b) illustrate this idea. Here we use a more advanced control technique called *Model Predictive Control* (MPC) which is an online optimal control algorithm. MPC will yield a better QoC than LQR because it, compared to LQR, takes advantages of a history of the previous control moves and solves an optimization problem over the *receding prediction horizon*, which is the time horizon where the algorithm will try to predict the plant states and to minimize the cost by yielding a update

control input path [59]. As can be seen in Fig 5.6(b), the difference between 20 ms and 30 ms, especially for S_1 and S_3 is very little.

It is straightforward to compute an optimal period, in terms of the mean TAAF, for a particular control algorithm. Increasing the control period will reduce the runtime computational burden for a single copy of the control task. However, it will also lead to a reduced size of S^3, and therefore all of the sub-spaces. In such a case, more time will be spent using higher level of fault-tolerance (i.e., more than one copy of the task). Fig 5.7 shows the optimal period around 30 ms, with the lowest average TAAF.

Fig 5.8 shows the comparison of sub-spaces using the different LQR and MPC algorithms. It clearly shows that a smaller control period will reduce the difference between the two algorithms. Since smaller control period yields a better QoC, it again shows that with a better quality of control, the impact of using a worse version on the sub-spaces will be reduced.

5.2 Case 2: Anti-lock Braking System (ABS) in a Straight Trajectory

For the case-study of an ABS in a straight line the canonical seven degree-of-freedom car model was used [16][42], along with the Dugoff tire model [17]. The user inputs to AdaFT are the physical plant dynamics and the SSC. The car dynamics are nonlinear; the reader should see [16][17][42] for detailed explanations of the car state equations and control algorithms. These equations allow us to implement the system using AdaFT API, similar to the template shown in Section 4.3.1. Below is the summary of the setup.

1. Actuator noise standard deviation is 5 N, and the mean is the true value obtained from the control task.

2. Sensor noise standard deviations for the two states (vehicle speed, wheel speed) are: (0.5 m/s, 0.5 m/s).

3. Initial condition of the two states is: (30 m/s, 30 m/s)

4. Particle filter is used for filtering out sensor noises.

5. ABS control task period: 20 ms; deadline 20 ms; WCET: 2 ms

5.2.1 Simulation Setup

Denote the front wheel steering angle as δ. Denote the longitudinal tire forces at the front left, front right, rear left and rear right tires as $F_{xfl}, F_{xfr}, F_{xrl}, F_{xrr}$, respectively. Denote the lateral forces at the front left, front right, rear left and rear right tires as $F_{yfl}, F_{yfr}, F_{yrl}, F_{yrr}$, respectively [42].

$$m\ddot{x} = (F_{xfl} + F_{xfr})cos(\delta) + F_{xrl} + F_{xrr} - (F_{yfl} + F_{yfr})sin(\delta) + m\dot{\phi}\dot{y} \tag{5.5}$$

$$m\ddot{y} = F_{yrl} + F_{yrr} + (F_{xfl} + F_{xfr})sin(\delta) + (F_{yfl} + F_{yfr})cos(\delta) - m\dot{\phi}\dot{x} \tag{5.6}$$

$$I_Z\ddot{\phi} = l_f(F_{xfl}+F_{xfr})sin(\delta)+l_f(F_{yfl}+F_{yfr})cos(\delta)-l_r(F_{yrl}+F_{yrr})+\frac{l_w}{2}(F_{xfr}-F_{xfl})cos(\delta)+ \frac{l_w}{2}(F_{xrr} - F_{xrl}) + \frac{l_w}{2}(F_{yfl} - F_{yfr})sin(\delta) \tag{5.7}$$

where x, y, ϕ are vehicle longitudinal, lateral position and orientation, respectively. And the lengths l_f, l_r and l_w refer to the longitudinal distance from the center to the front wheels, longitudinal distance from the c.g. to the rear wheels and the lateral distance between left and right wheels (track width), respectively.

Define the slip angles at the front and rear tires as follows:

$$\alpha_f = \delta - tan^{-1}(\frac{\dot{y} + l_f\dot{\phi}}{\dot{x}}) \tag{5.8}$$

$$\alpha_r = -tan^{-1}(\frac{\dot{y} - l_f\dot{\phi}}{\dot{x}}) \tag{5.9}$$

Define the longitudinal slip ratios at each of the four wheels using the following equations, where r_{eff} is the effective radius of the wheel, and ω_w is the angular velocity of the wheel:

$$\sigma_x = \frac{r_{eff}\omega_w - \dot{x}}{\dot{x}}, during\ braking \tag{5.10}$$

$$\sigma_x = \frac{r_{eff}\omega_w - \dot{x}}{r_{eff}\omega_w}, during\ acceleration \tag{5.11}$$

We used the Dugoff tire model [17] to calculate the tire forces, since it can be used separately in longitudinal and lateral directions even for a combined lateral-longitudinal tire model.

Denote the cornering stiffness of each tire as C_α and longitudinal tire stiffness as C_σ. Then the longitudinal force of each tire is given by:

$$F_x = C_\sigma \frac{\sigma}{1+\sigma} f(\lambda) \tag{5.12}$$

and the lateral tire force is given by:

$$F_y = \frac{C_\alpha (tan(\alpha))}{(1+\sigma))} f(\lambda) \tag{5.13}$$

where λ is given by:

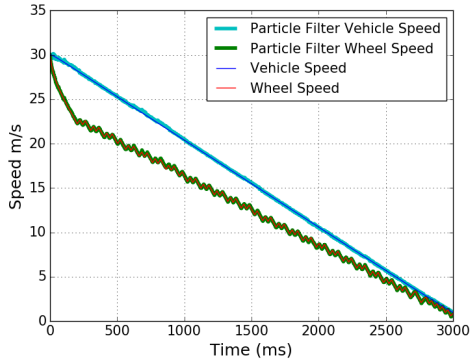
$$\lambda = \frac{\mu \times F_Z(1+\sigma)}{2\sqrt{((C_\sigma\sigma)^2 + (C_\alpha \times tan(\alpha))^2)}} \tag{5.14}$$

And

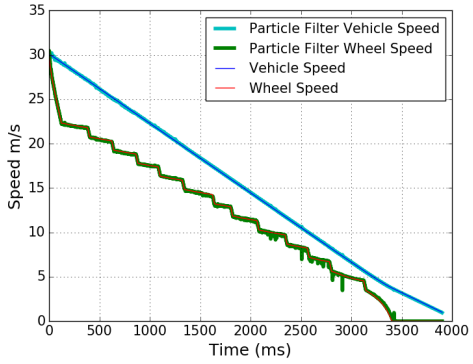
$$f(\lambda) = (2-\lambda)\lambda, if\ \lambda < 1 \tag{5.15}$$

$$f(\lambda) = 1, otherwise \tag{5.16}$$

Based on the Dugoff tire model, an optimal point of the slip ratio would give the maximum possible friction force during braking. This optimal point is typically within the range of 0.1 to 0.25, which might be different for each tire model and could be obtained offline.



(a) Road Friction Coefficient of 1.0

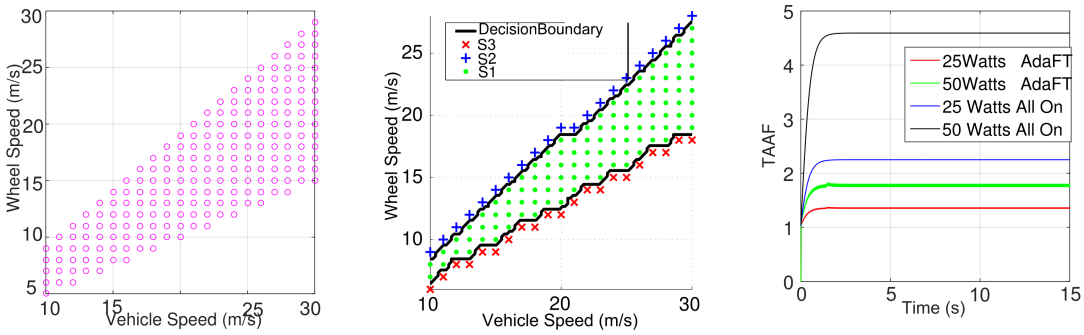


(b) Road Friction Coefficient of 0.8

Figure 5.9. ABS in a Straight Line. Here we show the state trajectories with different road conditions. The Particle Filter is used for filtering out sensor noises, and it shows a perfect inference here.

There are many algorithms to control the brake pressures to remain within this optimal region. We experimented with the canonical PID control algorithm for the simulation. We use the difference between the optimal point 0.2 and the actual slip ratio as the cost function to estimate the worst-case wrong control input.

The vehicle's SSC was generated by modeling the vehicle as a point mass and using the standard Dugoff tire model [17]. Under the assumption that the vehicle mass is 1,500 kg, the initial velocity is 30 m/s and the safe stop distance is 55 meters, the generated SSC is shown in Figures 5.10a and 5.11a for two road conditions: dry and wet, respectively. These two figures mean that as long as the longitudinal slip ratio, defined in Equations 5.10 and 5.11, is within this region, it is safe and can guarantee a safe stop distance. Note that with different initial conditions and safe stop distance requirements, the SSC region will be different. Accurately determining the SSC is beyond the scope of this dissertation, and is the responsibility of the domain specialist.



(a) SSC (b) FT Sub-spaces (c) TAAF

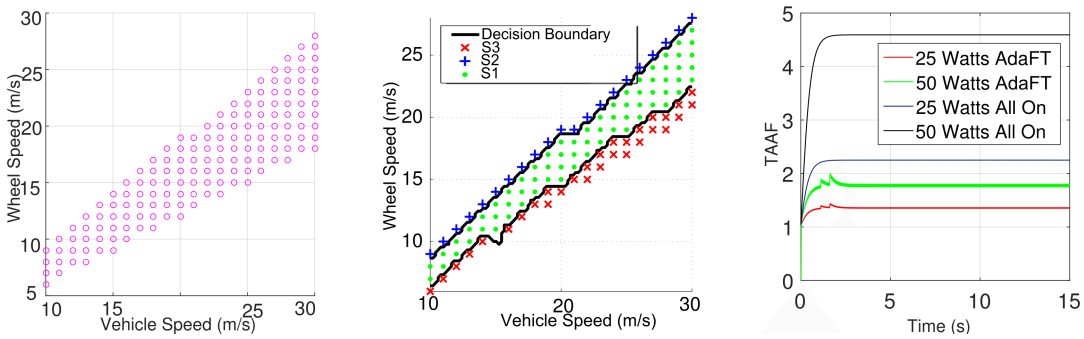
Figure 5.10. Sub-spaces and TAAF of ABS in a Straight Line with a Road Friction Coefficient of 1.0. (a) shows the SSC, inside which it is sufficient for a safe stop if the control inputs are correct. (b) shows the sub-spaces and the decision boundaries using our fitted machine learning algorithm. (c) shows TAAFs benefits.

5.2.2 Results

Figure 5.9 shows the simulation results regarding the state trajectories under two road friction conditions. Note how well the Particle filter tracks the system states, and that, as expected, for a worse road condition the total time for a complete stop will increase.

Since this is a two-dimensional system, a scatter plot with the corresponding decision boundary determined from the machine learning classification step can be easily plotted. AdaFT generates the sub-spaces shown in Figures 5.10b and 5.11b. Note that the gains from using adaptive fault-tolerance: almost 50% of the time, the vehicle is in the S_1 sub-space, meaning that the computational workload is significantly reduced.

Using machine learning the sub-spaces were expressed as a lookup table. Since this system only has two dimensions, and from Figures 5.10(b) and 5.11(b) there are clear linear boundaries between the sub-spaces, therefore even a linear machine learning algorithm such as logistic regression can perform very well. We include results from three fitted algorithms: support vector machine with a Gaussian (rbf) kernel, logistic regression and neural network. The lookup table had 138, 15, and 93 trained parameters for each algo-



(a) SSC (b) FT Sub-spaces (c) TAAF

Figure 5.11. Sub Spaces and TAAF of ABS in a Straight Line with a Road Friction Coefficient of 0.8. Similar information as compared to Fig 5.10 is shown here, except that this is for a slightly worse road condition.

Table 5.3. Comparison of Learning Algorithms for ABS

	LR	*NN*	*SVM*
Number of Trained Parameters	15	93	138
Testing Accuracy	100%	100%	100%

rithm, respectively. All three algorithms achieved 100% testing accuracy, and obviously, the precision and recall can both be 100% since our accuracy is 100%.

To calculate the TAAF, we experimented with similar power usage levels as in the first case study: 25 watts and 50 watts. Figures 5.10c and 5.11c show the improvement of TAAF. Again note the reliability improvement from using adaptive fault-tolerance over the standard redundancy-everytime approach.

We now discuss about the impact of environmental conditions on the sub-spaces. Intuitively, as the road condition worsens, i.e., the friction coefficient become smaller, the road will become more slippery. Given a fixed required safe stop distance of 55 meters and initial vehicle speed of 30 m/s, the sub-spaces will be significantly impacted by the worse road conditions. Fig 5.12 illustrates this. For a road condition below 0.6 friction coefficient, S_1 is empty, which means that for a worst control input, even for a single control period, the state will fall out of S^3; the control inputs should be always correct for a safe stop. For

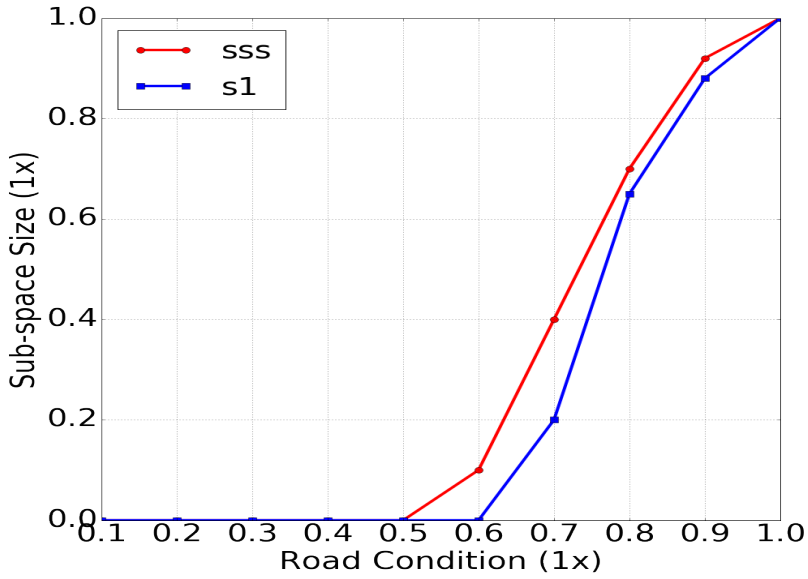


Figure 5.12. Sub Spaces Sizes with Different Road Conditions. The baseline is a road condition of 1.0. The two curves show the fraction of S^3 and S_1 compared to the baseline.

road condition below 0.5, S^3 reduces to zero, which means if the road is too slippery, an initial vehicle speed of 30 m/s (corresponding to about 65 mph) will never be able to stop within 55 meters. As can be seen in Fig 5.13, the final stop distance under this condition is about 60 meters, even with perfect control at every time step.

It should be noted that for this ABS case study, the required safe stop distance, the road condition and the initial vehicle speed all impact the sub-space sizes. Therefore, in order to handle different scenarios during the runtime execution, it would make more sense to include all these three variables in the ”feature” list, in addition to the original two features, when generating the sub-space and performing the offline training. As a rule of thumb, all variables that might change during operation and that might affect the sub-space sizes should be treated as the ”pseudo” state components included in the feature list.

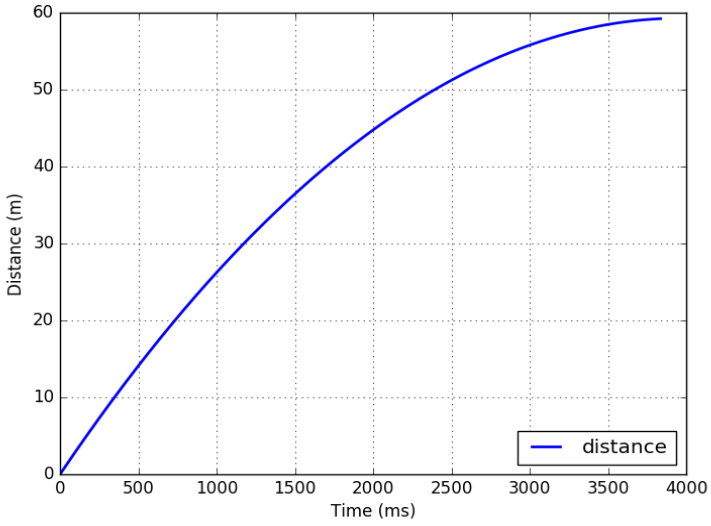


Figure 5.13. Distance of ABS-equipped Vehicle under the Road Coefficient of 0.5. It clearly shows that even with correct control inputs, the final stop distance is about 60 meters.

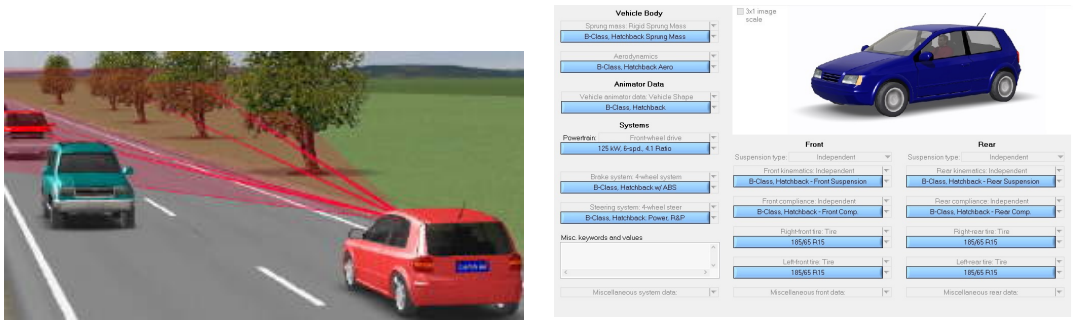
5.3 Case 3: Automated Highway System in a Platoon

Grouping vehicles into platoons is an approach to increase highway capacity. An automated highway system (AHS) is one technology for doing this [4].

Platoons decrease the distances between cars using coupled control approaches, which coordinate acceleration and braking among cars. Such synchronization allows for a considerable increase in traffic throughput. An overview of AHS can be found in [4]. Cars in a platoon are an example of a distributed CPS, with numerous cooperating participants.

5.3.1 Simulation Setup

As seen before, AdaFT requires the user to provide functions to update the controlled plant state and to generate the control signal. The physical model is much more complex than a straight line ABS system, therefore Carsim [48] integrated with Matlab/Simulink was used for this purpose. Carsim, a commercial automotive simulation tool, provides realistic modeling of vehicles for a variety of road conditions; Matlab/Simulink is used



(a) Carsim for Multiple Vehicles (b) Carsim Vehicle Body

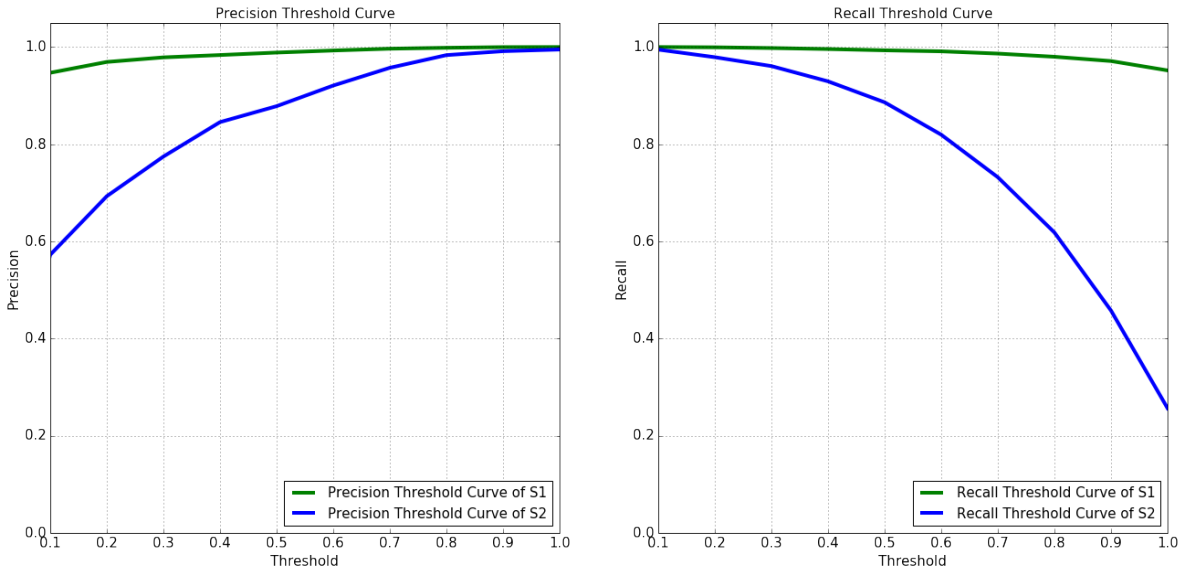
Figure 5.14. Carsim Platoon System

to generate the control signals and update the vehicle state appropriately. As shown in Figure 5.14, Carsim allows users to configure multiple cars interacting with each other. User could also specify the physical parameters of the vehicle body, and Carsim would generated sophisticated differential equations based on these parameters.

The platoon consists of a leader car and one or more followers. The state space vector of the platoon is $(v_\ell, \mathbf{v}_f, \mathbf{s}_f)$ where v_ℓ is the velocity of the platoon leader, $\mathbf{v}_f$ is the vector of velocities of the follower car(s), and $\mathbf{s}_f$ is the vector of spacing between each follower and the car in front of it.

The control task is the adaptive cruise control with the constant time-gap (CTG) policy [42]. The idea is to set a desired inter-vehicle spacing based on this constant time gap (e.g., 2 seconds). This desired distance then varies linearly with the relative velocity. The desired acceleration is computed by the formula: $\ddot{x}_{desired}(t) = -\frac{1}{h}(\dot{\epsilon}(t) + \beta(\epsilon(t) + h\dot{x}(t)))$, where h is a design parameter, β is the desired time gap, and ϵ is the inter-car distance. Details about this algorithm can be found in [42]. Obviously, a negative acceleration means braking.

As for the Safety Space Constraints, the minimum inter-car distance is set at 5 meters; anything less is defined as control failure and unsafe conditions.



(a) Precision Threshold Curve (b) Recall Threshold Curve

Figure 5.15. Precision Recall Trade-off.

5.3.2 Results

For the learning algorithm, the logistic regression performs poorly, while more complex algorithms are much better. After adjusting the proper prediction thresholds, the precision is close to 100%, as shown in Figure 5.15. Detailed plots of thermal damage to the processors also follow similar lines to those in the ABS case study and are omitted here.

We use a different perspective to analyze the results here. Figure 5.16 shows the sub-spaces for the first follower car. Figure 5.18a shows how to deal with the follower car joining the platoon and synchronizing its speed appropriately. Sub-spaces are shown for various speeds of the joining car for an inter-car distance of 40 meters. If its speed is low and it needs to accelerate to join the platoon, then little, if any, fault-tolerance is needed; if its speed is higher, the required fault-tolerance increases. In Figure 5.18b, we consider the impact of inter-car distance and platoon speed on the sub-spaces after speed synchronization has been established between the leader and the follower vehicles. As the inter-car distance decreases and/or the platoon speed increases, the highway throughput increases

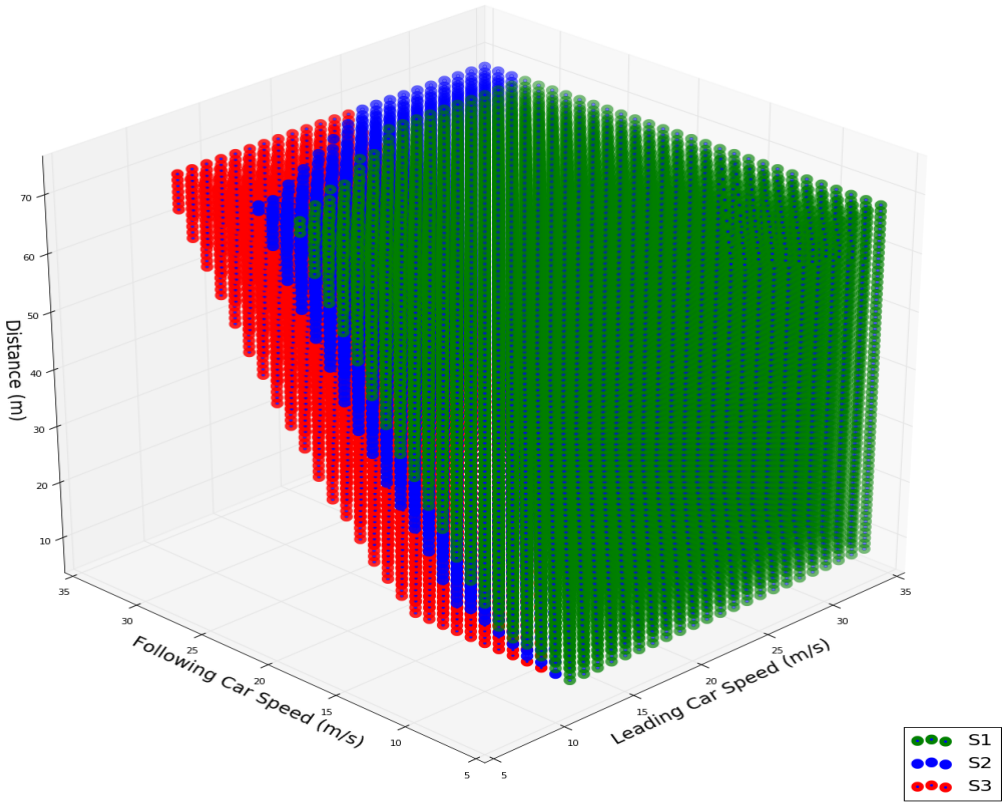


Figure 5.16. Platoon Follower 3D Sub-spaces for CTG Algorithm

(in terms of cars per time unit transiting at any given point) but the computational burden also increases for each car, since a higher level of fault-tolerance is needed. For example, at a speed of 25 meters/sec, an inter-car distance of less than 21 meters will require fault-correction; between 22 and 34 meters, fault-detection is sufficient, and above 34 meters, no fault-tolerance is needed.

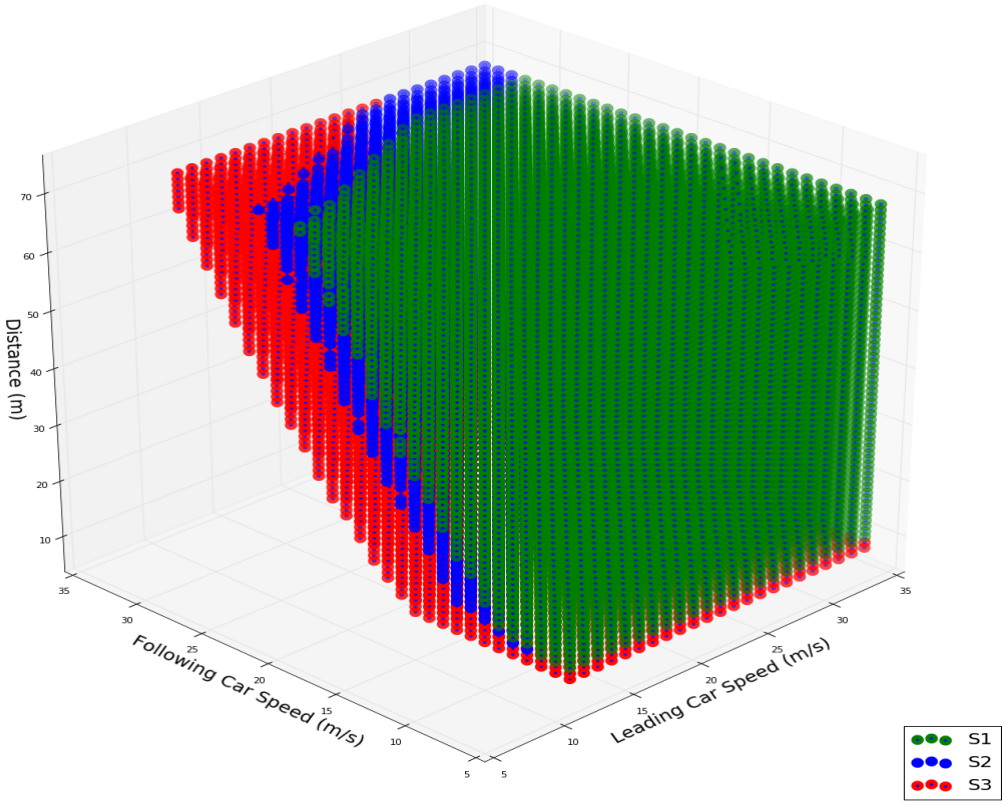
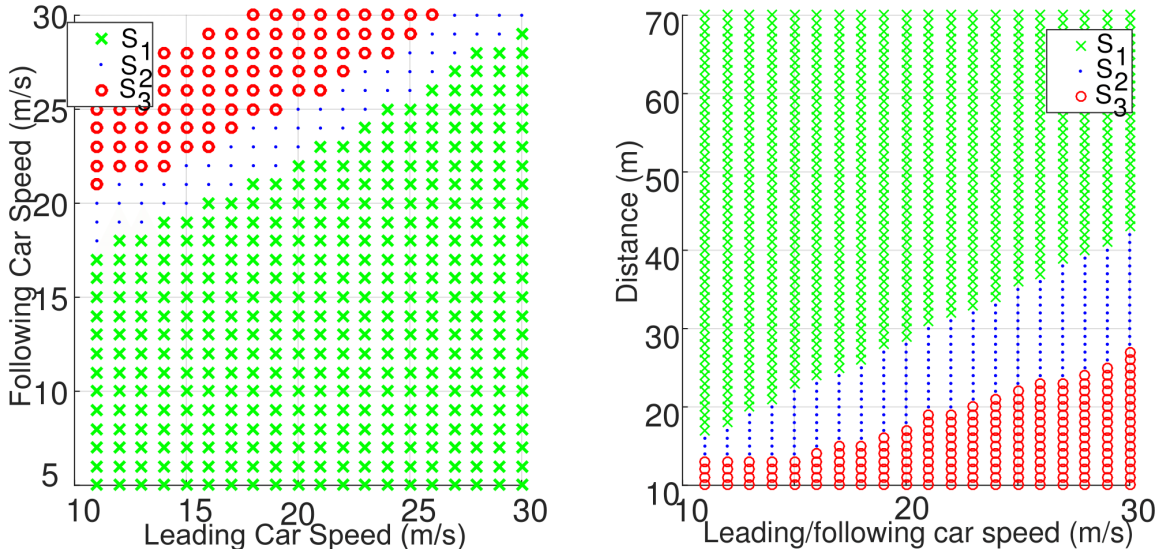


Figure 5.17. Platoon Follower 3D Sub-spaces for CTG Algorithm with Actuator Noise. Here the noise level is set as zero mean, and 10% standard deviation.



(a) Distance Fixed at 40 Meters

(b) Leading car speed same as follower

Figure 5.18. Platoon Follower Car Sub-spaces. Here we fix certain variables for the visualization purpose.

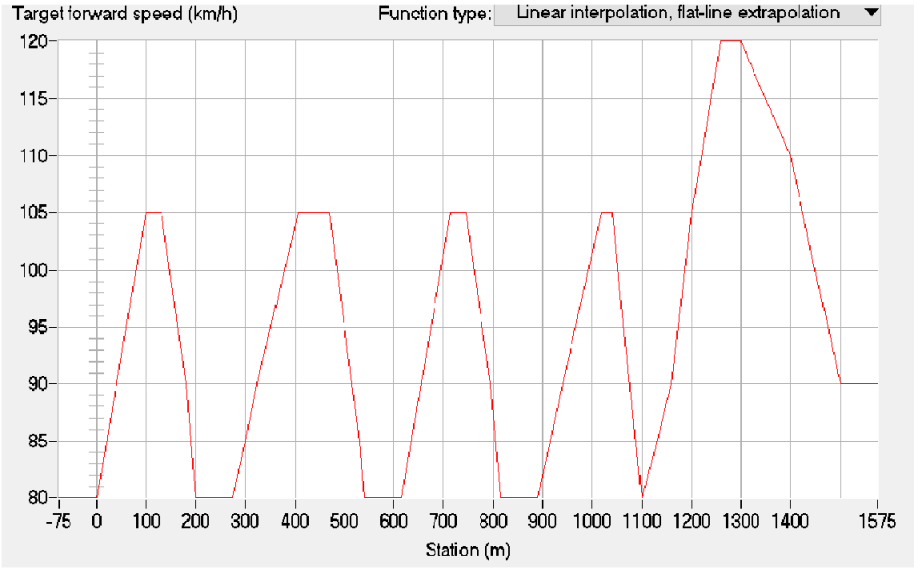


Figure 5.19. Platoon Leading Car Speed Pattern. X axis is the location of the car, and y axis is the speed the car will perform.

We now extend our model to four cars, one leading car and three following cars. The leading car is performing a sinusoidal driving pattern (i.e., its speed follows a sine wave). We also introduce multiple versions of the control tasks, with the complex version (version 1) as the constant time-gap (CTG) algorithm as before, and the simple version(version 2) as the simple PID control. We claimed in Section 4.2.8 that for multiple-version multiple-control system, it might be preferable sometimes to use the most conservative S^3; that is, to generate S^3 based on the simplest version for each control task, due to high computational burden. However, for this case, it is a single-control system, with only one control task (the cruise control), and therefore it is reasonable to use a separate sub-space for each version of the task. As a result, we would have two different sets of sub-spaces for the CTG and PID versions of the task. During runtime, the load tuner will first determine which version to run, then decide on the number of copies for that version.

Fig 5.20 shows the trade-off between TAAF and QoC, with different QoC constraints, which in this case is the desired distance between two cars. This desired distance can also be viewed as the average highway throughput. If this QoC constraint is set as 10 meters, the complex version of the task will be triggered more frequently than the simple version, and the actual average distance between two cars will be around 25 meters. Because the average distance is larger than the desired distance, the complex version will be used most of the time, thereby causing a larger average TAAF of about 1.20. As expected, if we relax the QoC constraint to a larger value, the simple version will be used more often. In fact, once the actual mean distance is greater than the desired distance, the QoC constraint is satisfied and the simple version will be used during most of the time.

Fig 5.21 shows the trade-off between TAAF and the control period. The desired distance is fixed as 15 meters for this case. The general pattern is similar, except that for periods greater than 35 ms, the TAAF will saturate around 1.05. This is because the period is much larger than the actual execution time, allowing the processors to be idle for most of the time. Even if the control algorithm is using the complex version for most of the time, as

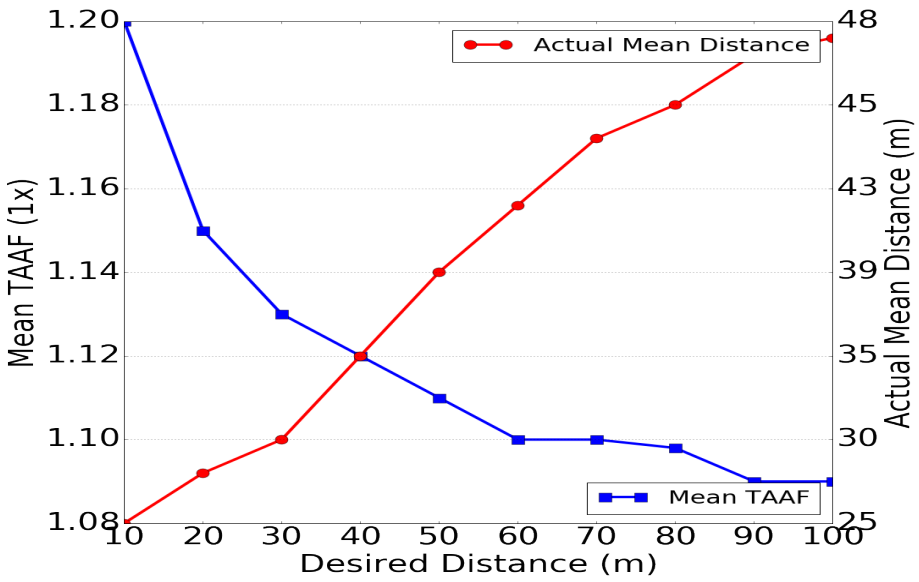


Figure 5.20. TAAF vs Desired Distance. The desired distance between two cars is the QoC constraint. Here we have two versions of the cruise control task, the complex version is the constant time-gap algorithm, while the simple version is the PID which will yield worse QoC. The period for both versions is 20 ms. We select a more reasonable power for the complex version as 10 watts, and for the simple version as 6 watts. The execution time for the complex version is set as 8 ms, while the simple version is 4 ms, which were measured using tools such as the Matlab's Embedded Coder Profiling with a hardware target (ARM Cortex A9)

the actual distance is larger than the desired distance, the running time of the processor will be small so that the TAAF will also be small. However, since the period keeps increasing, the control would be updated less frequently, and therefore the actual mean distance would still become larger.

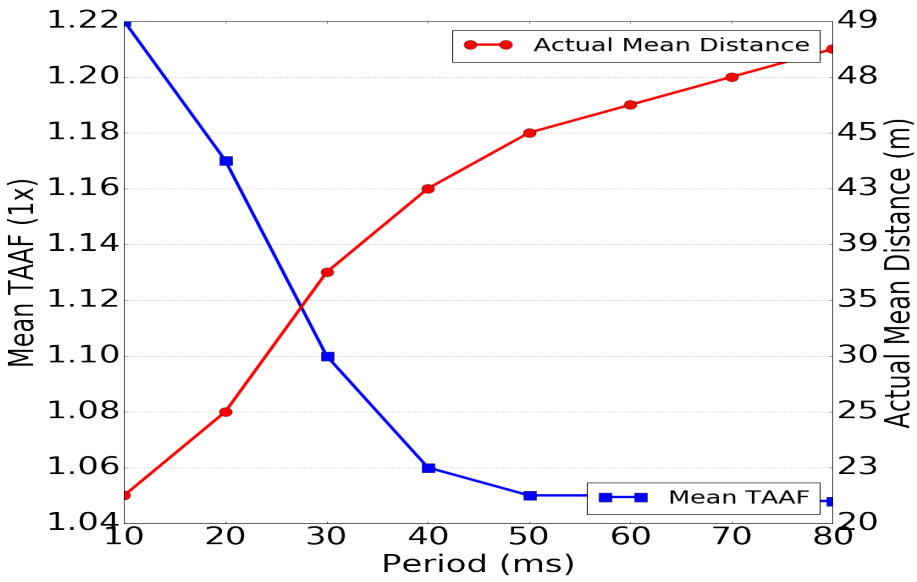


Figure 5.21. TAAF vs Period. Here the desired distance is fixed at 15 meters. All other configurations are the same as in Fig 5.20

5.4 Case 4: Humanoid Rigid-Body Robot

In this example we consider a simple model of a humanoid robot that is capable of balancing on its own [37]. We make several assumptions:

- The robot motion is limited to a 2D plane (i.e., leaning backward and forward).
- We only have three degrees of freedom: rotation at the ankle, knee, and hip.
- The forces generated by the muscles are modeled as ideal torques between the adjacent body segments.

5.4.1 Simulation Setup

Figure 5.22 shows the model and its parameters.

There are four reference frames. The blue lower leg reference frame, L, is attached to the foot by a pin joint at the ankle point A and rotates relative to the foot with an angle θ_1. The green upper leg reference frame, U, is attached to the lower leg by a pin joint at the knee point K and rotates relative to the lower leg with an angle θ_2. The red torso reference

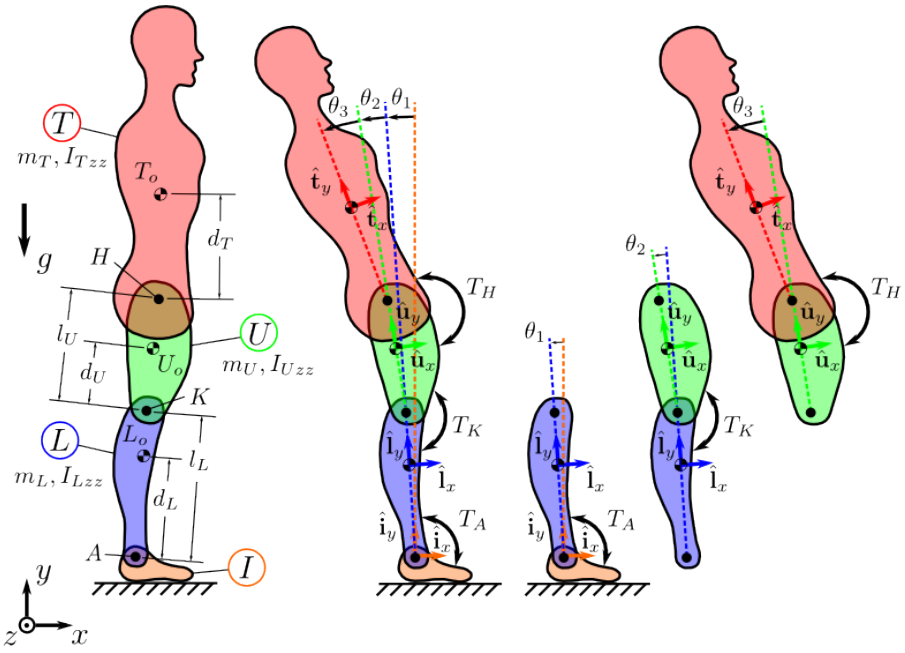


Figure 5.22. Humanoid Robot Balancing System [37]

frame, T, is pinned to the upper leg at the hip point, H, and rotates relative to the upper leg with an angle θ_3. Note that all rotations are along the z axis.

The three points L_o, U_o, T_o are the mass centers of the body segments. These are located on the line connecting the proximal and distal joints of each body segment.

Gravity is directed downwards (y axis) and applies a force with a magnitude of $m_L g$, $m_U g$, $m_T g$ at each mass center, respectively.

Three torques represent the forces from the muscle contraction. The ankle T_A, knee T_K, and hip T_H torques apply equal and opposite torques to the adjoining body segments.

We first list all the notations used: lower leg length: l_L, lower leg mass center distance: d_L, lower leg mass: m_L, lower leg inertia: I_{Lz}, upper leg length: l_U, upper leg mass center distance: d_U, upper leg mass: m_U, upper leg inertia: I_{Uz}, torso length: d_T, torso mass: m_T, torso inertia: I_{Tz}, gravity constant: g.

We show below the A and B matrices, and define the physical dynamics in the form of Equation 5.1, and plug the numerical values of the following constants into the equations: [l_L: $0.611m$, d_L: $0.387m$, m_L: $6.769kg$, I_{Lz}: $0.101kg * m^2$, l_U: $0.424m$, d_U: $0.193m$, m_U: $17.01kg$, I_{Uz}: $0.282kg * m^2$, d_T: $0.305m$, m_T: $32.44kg$, I_{Tz}: $1.485kg * m^2$, g: $9.81m/s^2$]. For the details how these matrices are derived, readers can refer to [37].

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 18.337 & -75.864 & 6.395 & 0 & 0 & 0 \\ -22.175 & 230.549 & -49.01 & 0 & 0 & 0 \\ 4.353 & -175.393 & 95.29 & 0 & 0 & 0 \end{bmatrix} \tag{5.17}$$

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0.292 & -0.785 & 0.558 \\ -0.785 & 2.457 & -2.178 \\ 0.558 & -2.178 & 2.601 \end{bmatrix} \tag{5.18}$$

We will demonstrate in this case study a multiple-control, multiple- version example. We use the LQR approach and the Q and R matrices to generate the control gain of K matrix. Recall from Case Study 1 that the Q matrix defines the cost of the states deviating from the desired ones. For this case study, we have the following setup for the control tasks:

- There are three control tasks controlling three rates, which affect three joint angles. These three tasks are denoted by u_1, u_2 and u_3.
- For version 1 (the better version) we use the LQR algorithm; for version 2, we use a simple *pole placement* algorithm for a typical LTI state feedback system [59].

Table 5.4. Examples of Worst Case Control Vectors

Angle 1	Angle 2	Angle 3	Rate 1	Rate 2	Rate 3	Control 1	Control 2	Control 3
0.087	-0.087	0.087	-0.087	-0.087	-0.087	250	-250	250
-0.65	-0.087	0.087	-0.087	-0.087	-0.087	-250	250	-250
0.65	-0.087	0.087	-0.087	-0.087	-0.087	250	-250	250

- All Three control tasks have the same period of 20 ms, which are tested to ensure the system safety and stability.
- Version 1 of each task would have 10 watts of power consumption. Version 2 would have 6 watts. In this case study, we use *McPAT* to estimate the power consumption [30].
- The WCET for each control task version 1 is 8 ms, while for version 2 is 2 ms. We use Matlab's Embedded Coder profiling tools to estimate the task execution time.

We also define the SSC to be:$-0.78 < \theta_1 < 0.78$, which means that as long as the lower leg angle is from -45 to 45 degrees, the robot is safe and can always balance itself.

As for the cost function for the wrong controllers, we use the same cost function discussed in Section 4.2.2, and we define the actuator bounds as from -250 to 250 N. From the B matrix of the system we find that each control force is not only directly applied at one rate, but also affects the other two rates as well. Therefore, an optimization scheme is used to find the worst combination of the control forces. Table 5.4 shows some examples of worst control forces given particular state vectors, where the angles are in $radians$ and the rates are in rad/s. We use the same numerical examples in [37], i.e., with 0.087 radians or 5 degree, 0.65 radians or 37 degree, etc.

5.4.2 Results

We use version 2 for each control task to generate S^3, since with version 2 for each control task the system would yield the worst QoC; all other version combinations would produce a better QoC. With version 2 for all the tasks AdaFT would generate the most

conservative S^3. On the other hand, if we use every version for each task to generate a corresponding S^3, the total size of S^3 would become intractable, especially when the number of versions for each task is larger than 2 (see Section 4.2.8).

Once we have generated the most conservative S^3, we can now generate the sub-spaces for each task. Recall that each task has its own set of sub-spaces. Particularly, for u_1, we use the worst case control vector to run the simulation for 20 ms, and label the initial points as in S_1 if the final state belongs to S^3. The same procedure is applied for u_2 and u_3. To obtain S_2 for u_1, we use zero control for u_1 and the worst controls for u_2 and u_3. After 20 ms, if the states are still inside S^3, the initial states belong to S_2. The same can be applied to u_2 and u_3.

θ_1 clearly has some relationship with the variable *level*, which indicates the fault-tolerance level 1, 2 or 3; but other variables don't show significant relationships. With a *Decision Tree* classifier, we have a training accuracy of 99.92%, and a testing accuracy of 97%. Table 5.5 shows the feature importance of each variable, and as expected, we see θ_1 with the highest importance of 0.34, and *rate1* also plays a significant role with an importance of 0.28. All other features are relatively unimportant, which is consistent with our intuition, since the only SSC we care about for this system is that θ_1 should be from 0 to 0.78 radians.

Recall that for the multiple control case, each control task will have its own sub-spaces, therefore each would have a corresponding classifier. All three classifiers have very similar results since the tasks have the same period. Our fitted machine learning algorithm has very similar performance results for the majority of the algorithms, with a total testing accuracy of 97%, except that the *Neural Network* with 3 hidden layers, each having 75 hidden units, and a regularization factor of 0.3, has a testing accuracy of 99.6%.

Table 5.5. Feature Importance of humanoid Robot System

Angle 1	Angle 2	Angle 3	Rate 1	Rate 2	Rate 3
0.34	0.13	0.06	0.28	0.108	0.057

Table 5.6. Accuracy and Prediction Time of humanoid Robot System

	Neural Network	Random Forest	Decision Tree
Testing Accuracy	99.6%	97%	97%
Prediction Time	3ms	1ms	0.0096ms

In order to apply our fitted algorithm to the system, we need to consider the trade-offs between the algorithm's accuracy and efficiency. Recall from the previous case studies that even for the shallow Neural Network with only one hidden layer, and a small number of hidden units, the testing accuracy is close to 100%. All other machine learning algorithms have similar performance. However, in this case, due to the high dimensionality and complexity, only a deep Neural Network achieves a high testing accuracy close to 100%. All other algorithms, after tuning the hyper-parameters and increasing the training data size, have the best testing accuracy around 97%. However, a deep Neural Network would require more parameters stored in memory and longer online prediction time. With the *ARM Cortex-A9* processor, we ran the Matlab/Simulink Embedded Coder to estimate the actual running time for a single prediction. Table 5.6 shows the testing accuracy and the prediction time for different algorithms. From the table, we see the advantage of using the *Decision Tree* classifier for runtime prediction with very little execution time overhead, and without much loss of accuracy.

We now show the simulation results of the QoC and TAAF. The initial condition we set for this simulation is [-0.45, 0.087, -0.087, -0.087, -0.087, -0.087] [37]. We define our QoC constraint as $-0.15\ rad < QoC < 0.15\ rad$, that is, θ_1 should be kept inside this range as much as possible. Figure 5.23 shows TAAFs for different combinations. In what follows we name each combination using the following pattern: Version_number_AdaFT(yes or no). For example, if a combination uses version 1 and AdaFT, we name it as *Version_1_AdaFT*. If a combination uses version 1 and 2 mixed together and always 3 copies of the CPU, we will call it *Version_Mix_All_On*.

In particular, it shows the TAAF difference for several version combinations, which clearly shows that with a higher power consumption, the benefits from AdaFT would be larger. Here version 1 has a power consumption of 10 watts, while version 2 has 6 watts. Therefore, the TAAF benefit from AdaFT for pure version 1 is the largest among all three cases. Note that for the TAAF *Version_Mix_All_On* and *Version_Mix_AdaFT* curves, initially version 1 is used, and after 1 second, version 2 is triggered, corresponding to the decreasing of the TAAF staring from 1 second, until converged.

Figure 5.23 also shows TAAFs for all on and AdaFT cases with all three versions (pure 1, pure 2, and mix) together. Intuitively, for both cases, version 1 and version 2 behave like upper and lower bounds, and the mixed version TAAF lies somewhere in between. Initially, θ_1 is 0.45 radians, which is larger than the QoC constraint of 0.15 radians, and therefore version 1 is triggered. After 1 second, version 1 is no longer needed, since the system has been balanced such that θ_1 is inside the QoC constraint region. Thus version 2 is activated until the end of the simulation.

If we include DVFS together with AdaFT's load tuning, we can achieve a lower TAAF. In AdaFT, each task has a *worst case execution time* (WCET) and a density function which generates a random value according to the density function of the actual execution time of the task. In our example we assume that the actual execution time of a task is always equal to half of its WCET. With such a setup and the DSR algorithm described in Section 4.2.10, we have the TAAF shown in Figure 5.23.

Figure 5.24 shows the number of copies of the control task u_1. As expected, using pure version 2 AdaFT will cause longer times when three copies are needed at the beginning of the simulation, due to its worst QoC output. Figure 5.25 to Figure 5.28 show the actual physical states trajectories of the system, together with the estimated Kalman filter results. Note that AdaFT and All On scheme will actually yield very similar physical trajectories since they are using the same control laws; the only difference will be the power consumption and TAAF. Again, using pure version 2 would cause worse QoC, in terms of the value

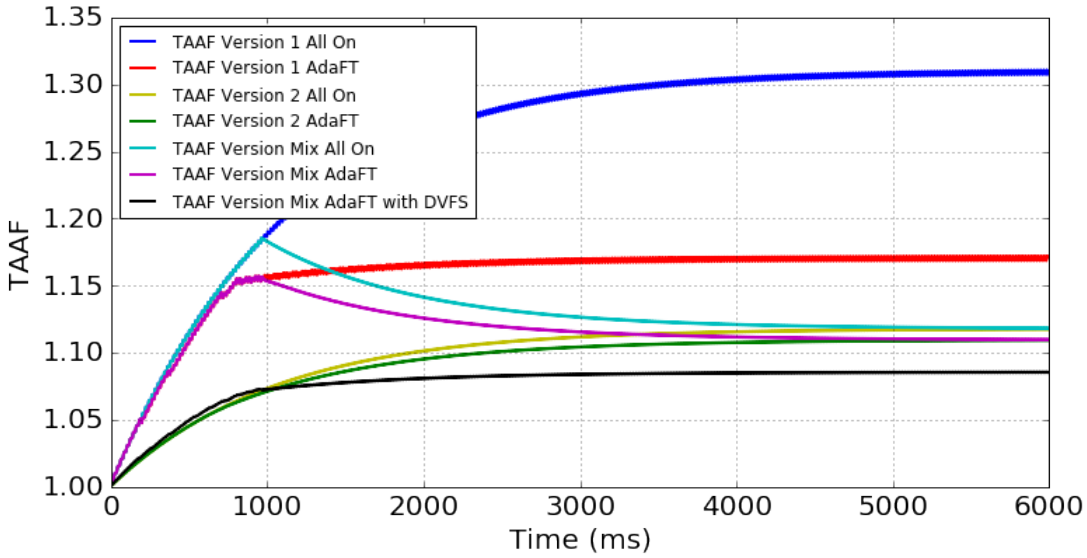
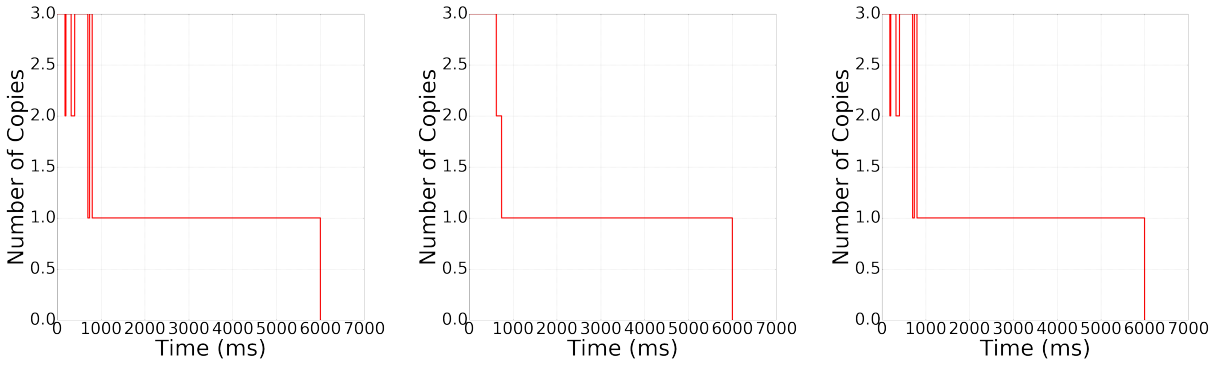


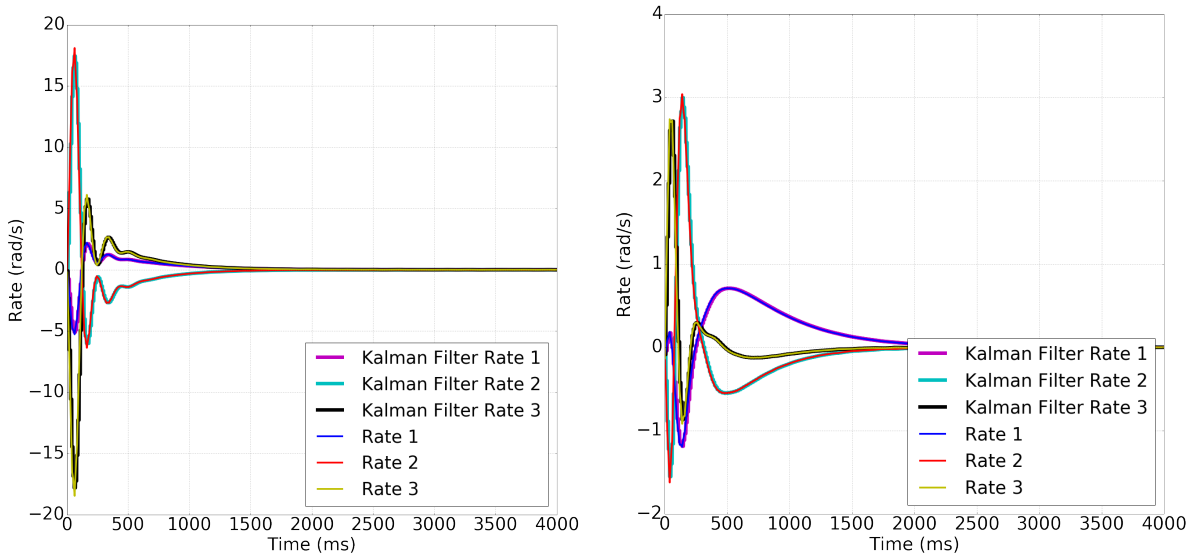
Figure 5.23. TAAF Of Humanoid Robot. Here we show all of the TAAFs in one figure for comparison. The power for version 1 is 10 watts, and for version 2 is 6 watts. The execution time for version 1 is 8 ms, and for version 2 is 2 ms. The period for both version is 20 ms.

of θ_1 at the beginning of the simulation, see Figure 5.25(b) and Figure 5.27(b). Pure version 1 has a best QoC (Figure 5.25(a) and Figure 5.27(a)), while the mixed version has a reasonably good QoC (Figure 5.26(a) and Figure 5.28(a)). The mixed version uses version 1 initially, and switches to version 2 after 1 second, and thus has a good balance between QoC and TAAF. Note that for the mixed version together with DVFS, we have a slightly worse QoC, as shown in Fig 5.28(b), since the execution time of the control tasks is increased due to the voltage scaling, which means the computational delay is larger. However, since the control period does not change, the QoC becomes slightly worse.



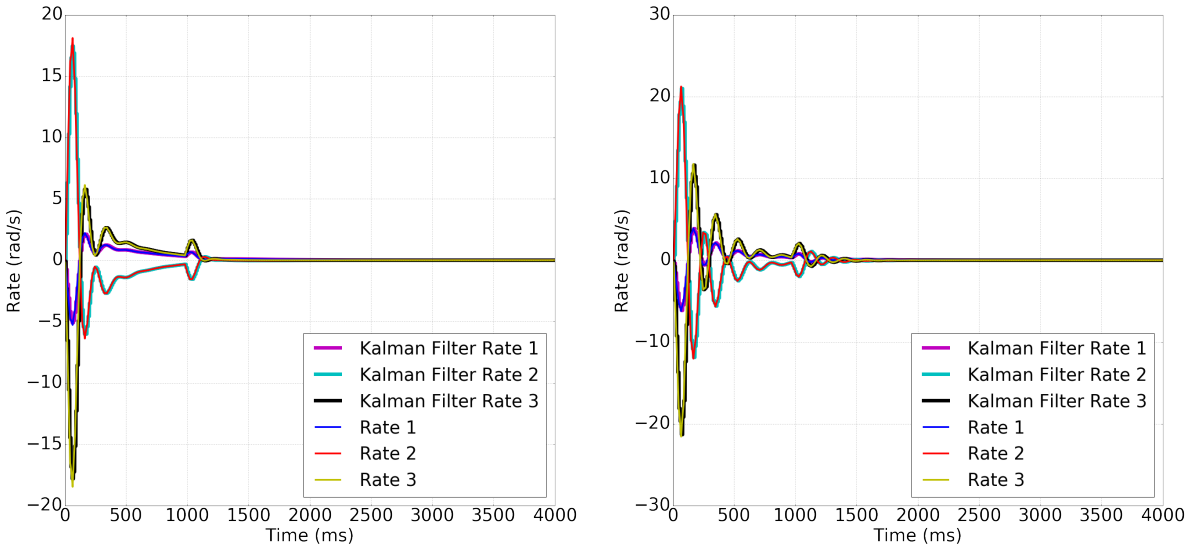
(a) Version 1 (b) Version 2 (c) Version Mix

Figure 5.24. Number of Copies of Control u_1 using AdaFT



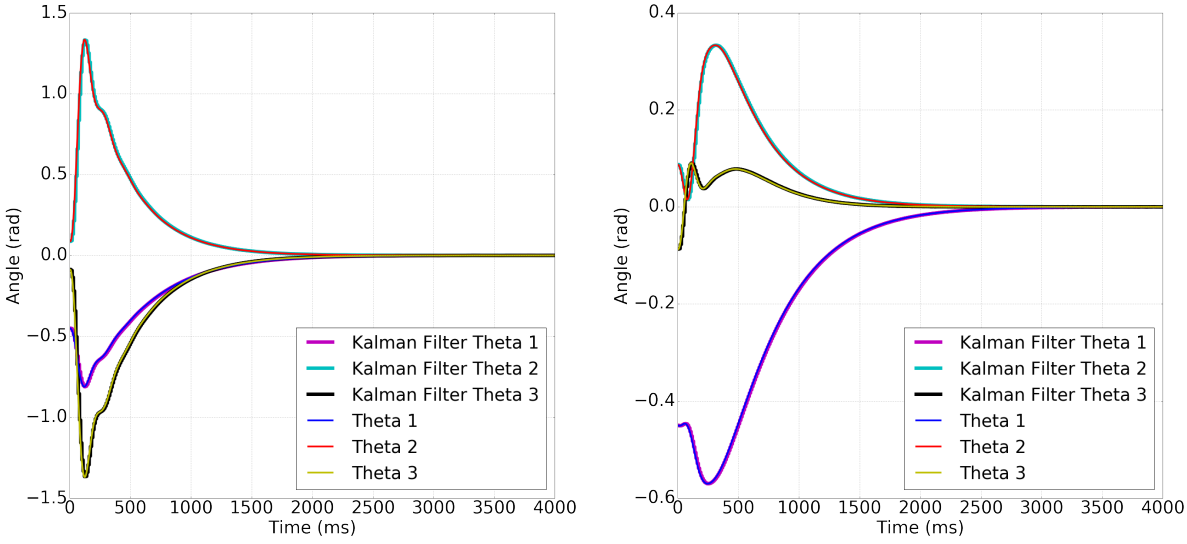
(a) Version 1 (b) Version 2

Figure 5.25. Humanoid Robot Rates using AdaFT or All On



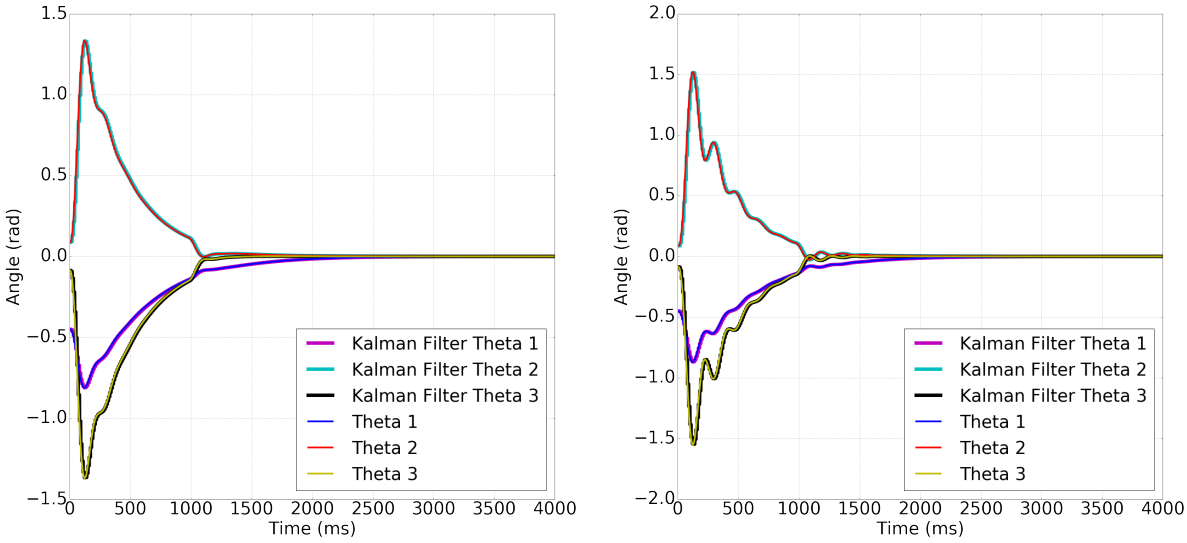
(a) Version Mix (b) Version Mix DVFS

Figure 5.26. Humanoid Robot Rates using AdaFT or All On



(a) Version 1 (b) Version 2

Figure 5.27. Humanoid Robot Angles using AdaFT or All On



(a) Version Mix

(b) Version Mix DVFS

Figure 5.28. Humanoid Robot Angles using AdaFT or All On

Now we discuss the hardware provisioning for this case study. For this configuration, the number of processors provisioned with the corresponding MTTF in years (which can be computed based on the integration of TAAF over time) is shown in Table 5.7, for the case of mixed version AdaFT with and without DVFS. The desired MTTF is set as 10 years, which means an unstressed processor has a 10 year expected lifetime. There are several conclusions can be made based on it. First, as expected, the higher the amount of the hardware that is available in the system, the higher the overall average MTTF or lifetime of the system will be. Second, additional TAAF benefits can be achieved with DVFS, which in turn would yield higher MTTF, given the same amount of hardware. Third, if there are many processors, the MTTF for both cases, with or without DVFS, will converge, due to the fact that each processor will be in the idle state for most of the time. Note that even in idle state, there is still an idle power of 0.5 watts, which is the reason for an MTTF of 9.55 years, rather than 10 years.

We now analyze MTTFs for different cyber-side configurations. We fix the power, actual execution time, and the WCET for version 1. The configurations include power consumption and the execution time to WCET ratio of version 2, the number of processors provisioned, and QoC constraints. Fig 5.29 shows the correlation matrix plot between various variables. In particular, it shows a strong positive correlation between the number of processors and all the MTTFs. However, the correlation between the number of processors and *MTTF_All_On* , and between the number of processors and *MTTF_DVFS* are weaker than that of the other two MTTFs. The intuition should be clear: using AdaFT the number

Table 5.7. MTTF In Years

	AdaFT Version Mix	AdaFT Version Mix DVFS
3 Processors	8.88 Years	9.16 Years
4 Processors	9.07 Years	9.28 Years
5 Processors	9.17 Years	9.34 Years
6 Processors	9.25 Years	9.38 Years
7 Processors	9.3 Years	9.42 Years
20 Processors	9.55 Years	9.55 Years

Table 5.8. Feature Importance and Performance of Random Forest for Different MTTF Scenarios

	Processors	QoC	Execution Time	WCET	Power	R^2 score
All On	0.374	0.062	0.265	0.0002	0.297	0.99
AdaFT	0.598	0.122	0.134	0.0002	0.145	0.99
DVFS	0.278	0.009	0.357	0.086	0.269	0.99
AdaFT DVFS	0.563	0.019	0.208	0.048	0.161	0.99

Table 5.9. Feature Importance and Performance of Random Forest for Different Energies Scenarios

	Processors	QoC	Execution Time	WCET	Power	R^2 score
All On	0.355	0.038	0.289	0.0001	0.316	0.99
AdaFT	0.626	0.086	0.134	0.0002	0.153	0.99
DVFS	0.269	0.005	0.364	0.088	0.272	0.99
AdaFT DVFS	0.577	0.015	0.201	0.046	0.159	0.99

of copies will be less than 3 for most of the time, while without AdaFT, the number of copies for the control tasks will always be 3 even for DVFS. Therefore, by increasing the number of processors, the MTTF benefits will increase faster for *MTTF_AdaFT* and *MTTF_AdaFT_DVFS*.

The QoC constraint seems to have very little correlation on all the MTTFs. The reason is intuitive: no matter what is the QoC constraint, the Robot system will only have its state values greater than this constraint at the beginning of the simulation, and it will converge to the balance position very quickly, as shown in Fig 5.27 and Fig 5.28. Thus, the QoC constraint in this case would have very little correlation with MTTF. Note that this is not always the case. For example, in the case study of Automated Highway Platoon system, since the leading car is performing a sinusoidal speed pattern, the actual distance between two following cars (the QoC in this case) will also change rapidly, with most of the time within the QoC constraint (the desired distance), and other times outside. For such cases, the QoC constraint will have stronger correlation between TAAFs and the corresponding MTTFs, as shown in Fig 5.20.

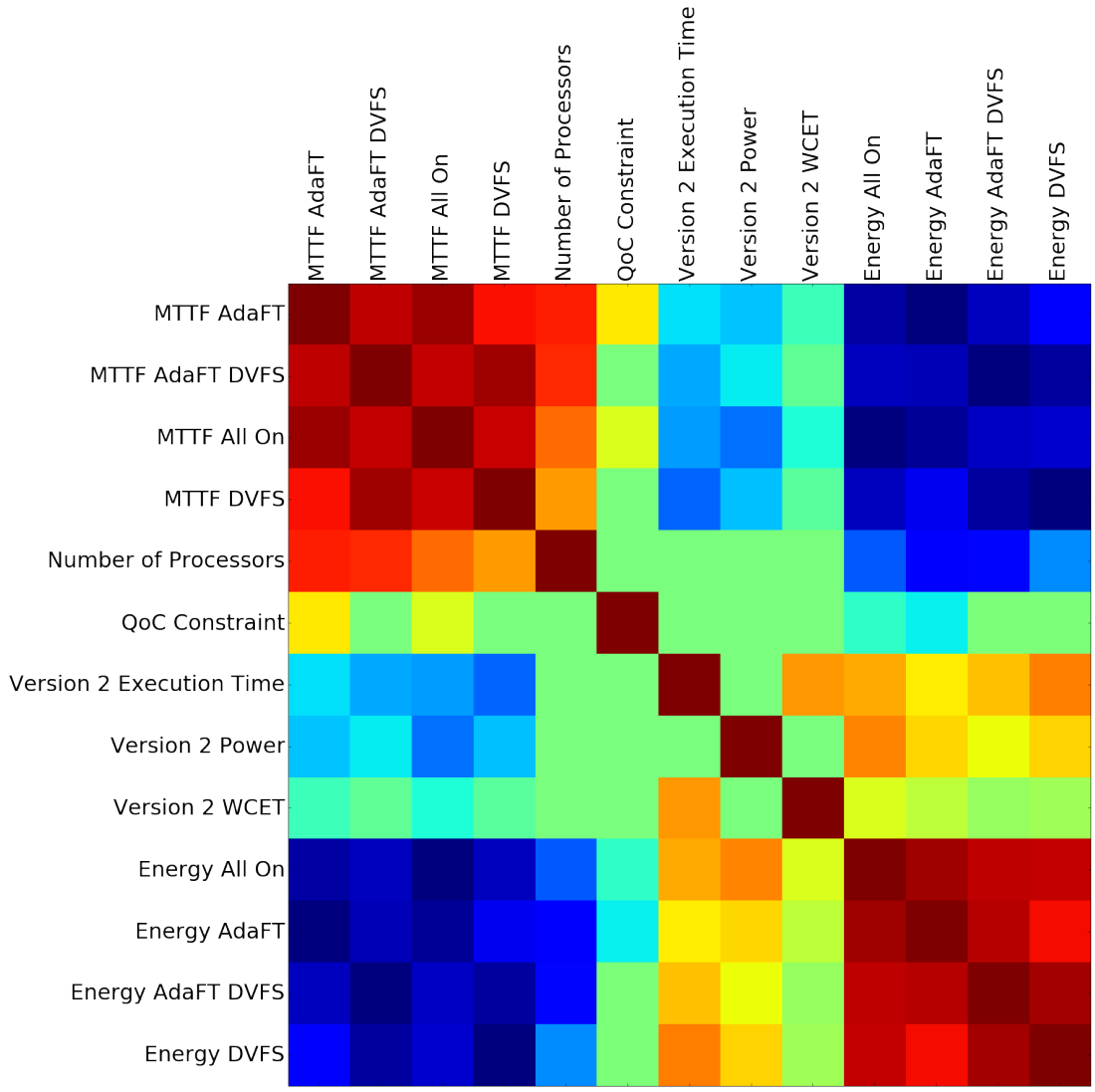


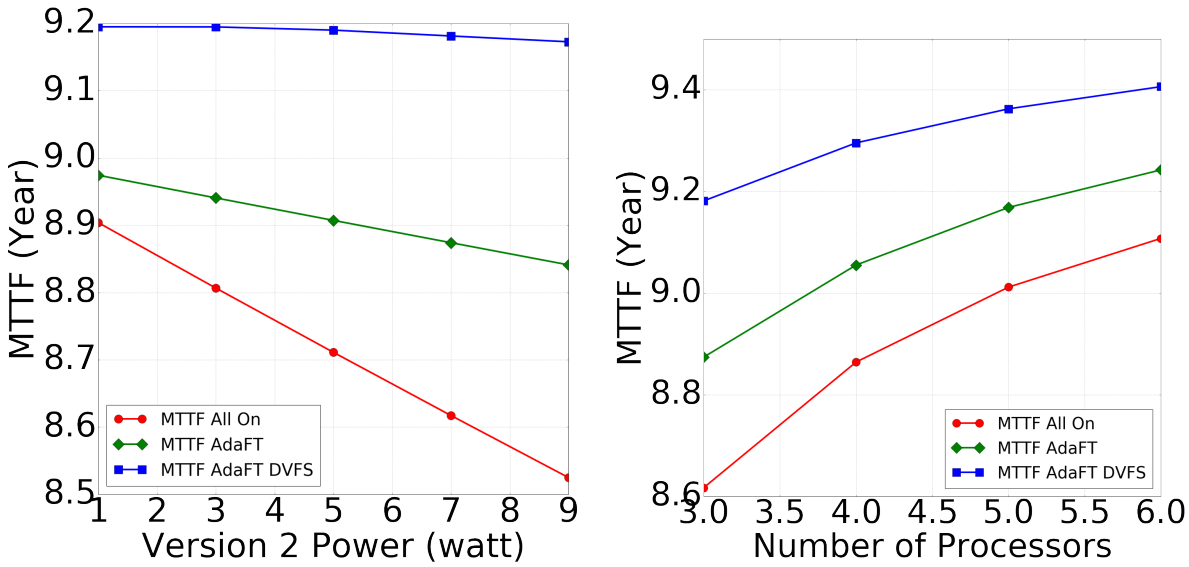
Figure 5.29. Correlation Matrix of MTTFs and Energy. Here a warm color means a positive correlation, and the cold color indicates a negative correlation. The more intense the color is, the stronger is the correlation between two variables

We next focus on the power consumption, which shows strong negative correlations. An interesting point is that the *MTTF_All_On* case shows the strongest correlation. The reason can be explained by intuition: since there are always three copies of the task running, if the power consumption of the task is reduced, the effect on *MTTF_All_On* would be the strongest. Note that here the *MTTF_DVFS* does not have a correlation as strong as that of *MTTF_All_On*. This is because by scaling the voltage, the power consumption of each task would also be scaled to a certain level. As was indicated in [12][13][60], there is a proportional relation between voltage and frequency ($f \propto v$), and power is proportional to voltage squared and frequency ($p \propto v^2 f$).

We will next discuss the execution time to WCET ratio. The execution time has a similar correlation between all MTTFs, while the WCET has less correlation between *MTTF_DVFS* and *MTTF_AdaFT_DVFS*. Thus, the execution time to WCET ratio would have a strong correlation between *MTTF_DVFS* and *MTTF_AdaFT_DVFS*. To explain this, recall that DVFS is based on the slack time after executing a control task, calculated as $slack = WCET - execution\_time$. The greater the slack, the more scaling the DFVS performs, thus resulting in a greater reduction of the power consumption of the actual control task.

From Fig 5.30 and 5.31 we can verify our discussions based on the correlation matrix plot Fig 5.29. In addition, Fig 5.30(a) shows that *MTTF_AdaFT* and *MTTF_AdaFT_DVFS* would have much less impact than *MTTF_All_On*, from the increase of the power consumption. Finally, Fig 5.31(a) also shows a "cross" point, around 3.5 ms, beyond which *MTTF_DVFS* performs better than *MTTF_AdaFT*. This means once the slack time is greater than a threshold, 0.3 ms in this case, the pure DVFS would behave better than pure AdaFT. Obviously, AdaFT together with DVFS would always perform the best.

It is interesting to use a machine learning algorithm to perform a regression based on such MTTF data set. If the regression algorithm has a good result, we might be able to predict the MTTF given arbitrary combinations of these parameters, without running any



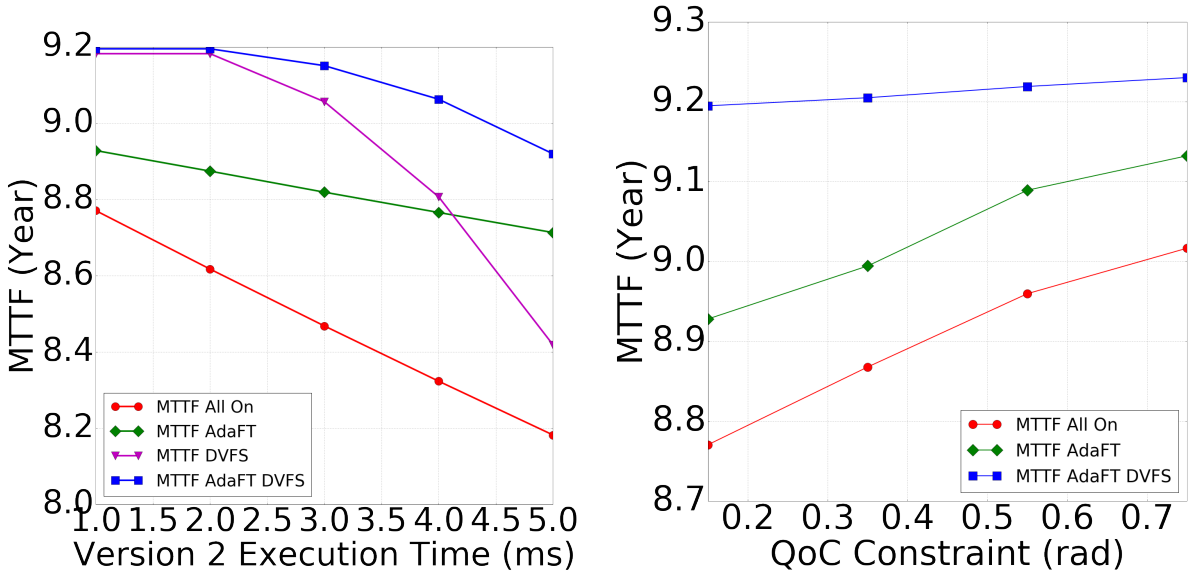
(a) Version 2 Powers

(b) Number of Processors

Figure 5.30. MTTF Comparison 1. Here we fix the WCET of version 2 as 4 ms, execution time as 2 ms, QoC constraint as 0.15 radians, and the number of processors as 3 for (a); all other parameters are the same except for version 2 power, which is fixed as 6 watts, for (b)

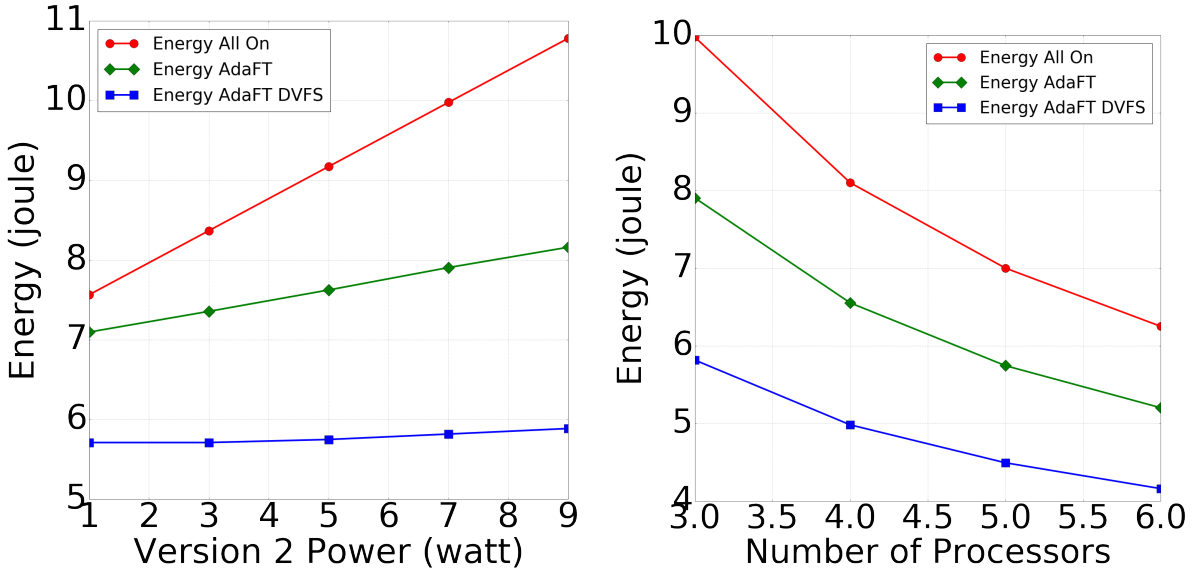
simulation. We use the Random Forest algorithm for the regression, and show the results in Table 5.8, which includes the feature importance and the final regression results in terms of the R^2 value. The feature importance is consistent with the correlation matrix plot in Fig 5.29, as well as Fig 5.30 and 5.31. The R^2 score shows a perfect accuracy for the algorithm.

Finally, we consider the energy consumption. Fig 5.29 shows the energy correlations with the five features. Fig 5.32 and 5.33 show the energy pattern as each feature changes. From these figures, we can see an inverse relationship compared to their MTTF counterparts. Therefore, the analysis holds for the energy comparison, except that the patterns are in inverse directions. Again, Table 5.9 shows a consistent feature importance matrix compared with the figures, and the R^2 score is high, indicating a good fit.



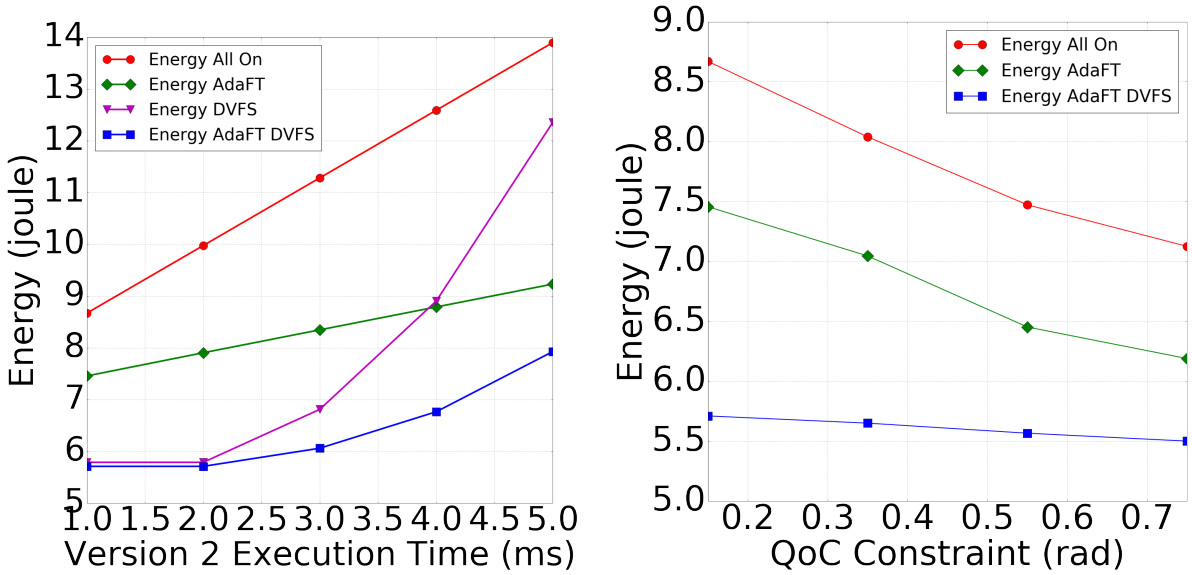
(a) Version 2 Execution Times (b) QoC Constraints

Figure 5.31. MTTF Comparison 2. Here we fix the WCET of version 2 as 4 ms, QoC constraint as 0.15 radians, and the number of processors as 3 for (a); all other parameters are the same except for version 2 execution time, which is fixed as 2 ms, for (b)



(a) Version 2 Powers (b) Number of Processors

Figure 5.32. Energy Comparison 1. Here we fix the WCET of version 2 as 4 ms, execution time as 2 ms, QoC constraint as 0.15 radians, and the number of processors as 3 for (a); all other parameters are the same except for version 2 power, which is fixed as 6 watts, for (b)



(a) Version 2 Execution Times

(b) QoC Constraints

Figure 5.33. Energy Comparison 2. Here we fix the WCET of version 2 as 4 ms, QoC constraint as 0.15 radians, and the number of processors as 3 for (a); all other parameters are the same except for version 2 execution time, which is fixed as 2 ms, for (b)

We now turn our attention to a more interesting case; we assign different periods for each control task. Specifically, we use 10 ms for u_1 and u_2 and 20 ms for u_3. Classification results, TAAF, MTTF, and energy consumption show similar patterns as before. One significant difference between this case and the previous one (with the same periods) is that the sub-space size for each task becomes significantly different from each other. The size of S_1 for u_1 and u_2 would be larger than that of u_3, since for u_3 we need to use the worst control vector for two times longer than u_1 and u_2. Since we are using the same S^3 (the most conservative S^3) for all three tasks, u_3's S_1 would become the smallest. The S_2 comparison is not as clear as S_1. The reason is that we always use the worst control vector to generate S_1, on the other hand, we use zero control for the relevant control task and the worst controls for the other control tasks, therefore the impact of zero control on the physical state component we care about (θ_1 in this case) with different periods from various control tasks will be different. Fig 5.34 shows the size difference. It should be noted that for the same period, only S_1 for all the control tasks will be the same, since they all use the worst case control vector. However, S_2 will be different, depending on how much impact a particular control task will have on the safety state component we care about (θ_1). For example, Fig 5.34 shows that even with a same period, u_2's S_2 is larger than u_1's S_2, which means u_2 will have less impact on θ_1.

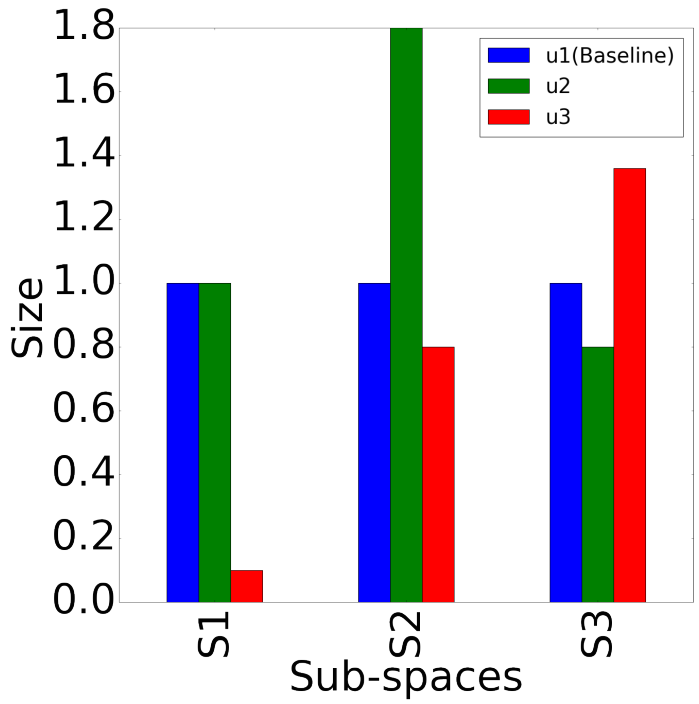


Figure 5.34. Subspace Size Comparison for Each of the Control Tasks. Here u_1 is set as the baseline.

CHAPTER 6

FUTURE WORK

6.1 Reinforcement Learning Algorithm

So far, we have taken advantage of the current physical side states to infer the level of fault-tolerance, as well as the version of each control task to be executed. Indeed, there exist other approaches that might further reduce the TAAF.

Recall that if the physical state is in S_1, we only need to run one single copy of the control task, since, by definition, even if the control task will yield a control signal that would cause the worst possible actuator command, the physical state would not leave S^3 before the next iteration of the task arrives. If we arbitrarily assign a control signal as the input to the actuator, this signal cannot be worse than the worst control signal; thus the actuator input will not lead the physical plant out of S^3. We may then conclude that we need zero copy of the control task to run at this time period! One possible approach to assigning this control signal is simply to use the one from the previous period. This is a greedy approach, however, that might incrementally pull the physical plant out of S_1. After certain periods, the physical plant might stay in S_3 that would need three copies of version 1 control tasks to run multiple periods in order to push the plant back into S_1. In such a case, the *long term* TAAF might, in fact, become worse than it was in our previous approach.

In order to improve the long-term TAAF, we can use the *reinforcement learning* approach. In particular, we will discuss how to incorporate *Q-learning* [54] into AdaFT, so that the long-term TAAF from AdaFT will indeed be improved.

Before we dive into the details of the design, we summarize the basic ideas behind the Q-learning algorithm. Q-learning is a model-free reinforcement learning technique.

Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) *Markov decision process* (MDP) [54]. The optimality is determined by a numerical variable called *Q value*, which is calculated based on the state and the action taken under this state. The Q value would give the expected utility of taking a chosen action in a particular state and following the optimal *policy* afterward. A *policy* is a set of rules that the agent (in the AI world, an agent is typically defined as the system that makes smart decisions) will follow when selecting actions, based on the given state it is in. After such an Q value function is learned, the agent will be able to select the optimal policy by simply selecting the actions which have the highest Q value in a particular state. One of the advantages of Q-learning is that many times it does not need a model of the environment, which is called the model-free algorithm. In addition, since Q-learning's Q value is the expected utility, it can handle problems with stochastic transitions and rewards, without any need for the adaptations. In fact, for any finite MDP, Q-learning would eventually find the optimal policy, which means the agent would be able to find an action with the maximum expected total reward over all successive steps, starting from the current state. For detailed mathematical equations and proofs, we refer the reader to [54].

We now discuss how to apply Q-learning to AdaFT. The most important steps of designing a Q-learning model include possible actions, state model, a reward function, and the hyper-parameters such as the learning rate, the discounted factor for the long-term reward. We first discuss the possible actions that the agent will explore to find the long-term reward, or the expected utility, after taking this action, given the state it is in. As mentioned before, if the physical plant is in S_1, we might want to assign a default value to the actuator so that zero copy of the control task will be actually executed. A reasonable choice of this default value of the control task is the one from last period, using the *zero-order hold* concept. Therefore, assume there are 2 versions of a control task, all possible actions can be listed as follows: (use version 1 with 3 copies, use version 1 with two copies, use version 1

with one copy, use version 2 with three copies, use version 2 with two copies, use version 2 with one copy, use previous value).

The state model in Q-learning is clearly different from the physical states of the plant. The major difference is that the physical plant states are typically continuous, whereas the Q-learning state requires that the state is finite. Additionally, the Q-learning state should reflect somehow the long-term reward of the system, which in our case is the long-term TAAF. Finally, the number of states should be kept reasonably small; otherwise, the Q-learning algorithm is intractable [54]. To handle the first issue, we can use a discrete representation of the physical states. For example, we can divide the continuous physical states into several sections, and check which section the current physical state is in at each update. For the second issue, a reasonable approach is to take the current TAAF value directly from the cyber side. Since TAAF is also a continuous value, we can use it the same way as we did the physical states.

Finally, to keep the total number of state variables small, we will use the sub-spaces and the QoC constraint to represent the current physical side information. We will use two variables – sub-space and QoC distance – to represent the physical side information. Let's first discuss the QoC distance. Take the inverted pendulum as an example, and assume that the QoC constraint is that the angle of the pendulum should be no larger than 0.15 radians. We will divide this value into, for example, six sections: smaller than -0.15, -0.15 to -0.075, -0.075 to 0, 0 to 0.075, 0.075 to 0.15, and larger than 0.15, which correspond to the value of -3, -2, -1, 1, 2, 3, respectively. If at some point, the angle of the pendulum is 0.1, then the QoC distance will be 1, since it is in the region of 0 to 0.075. For the sub-space, we can take the current sub-space of the plant and assign this value to this variable.

In summary, the state model will consist of three states: QoC distance, sub-space, TAAF. This state model is discrete and represents both the physical side information and the TAAF situation from the cyber side. Finally, this model has only three states, which will make the Q-learning procedure converge within a reasonable amount of time.

It should be noted that the Q value, once learned and converged, given a state and an action is a long-term reward. This Q value is updated at each step based on Equation 6.1, where R_{t+1} is the immediate reward observed after performing a_t in s_t, α is the learning rate, γ is the discount factor which indicates how much do we care about the future or long-term reward (a 0 means we only care about the immediate reward).

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \tag{6.1}$$

From Equation 6.1 we note that the update of the Q value is influenced by the immediate reward R_t. Thus we need to introduce such a reward function into AdaFT. The goal of AdaFT is to first to guarantee the system safety, then to satisfy the QoC constraint, and finally to keep the long-term TAAF as small as possible. We can assign a high negative reward if we observe, after taking an action that the system is in S_3; we can assign a medium negative reward if we observe the sub-space is S_2. For that matter, we can assign a small negative reward if we observe the physical plant is outside of the QoC constraint. Finally, we can assign a positive reward if the TAAF is below a certain threshold.

As for the hyper-parameters such as the learning rate and the discount factor, these are application- specific, as we will show in our case studies.

Fig 6.1 shows a sample Q-learning performance in the Humanoid Robot case study. It shows an even further reduction of TAAF than AdaFT with Mixed Version. An explanation for this is that, since we did not deliberately inject faults during the simulation, thus, as long as the control task is actually executed, its output is always correct. Therefore, every time the agent is in S_2, it sees no harm in just using one copy of the control task as in S_1, since the task is triggered for both cases; the agent will always use one copy even if it is in S_2. Another reason is that, at certain times in S_1, the agent may decide that it can have a higher long-term reward by using zero-order hold value from the previous task, with zero copy. This way, the TAAF can be significantly reduced.

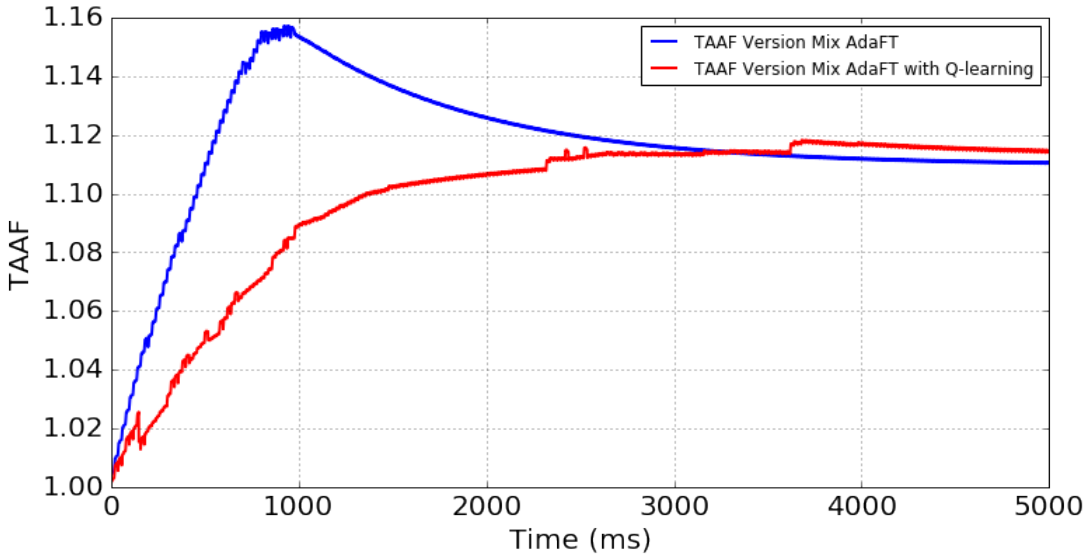


Figure 6.1. TAAF Of Humanoid Robot with/without Q-learning

One way to make TAAF worse is to manually inject faults at random times. Since the Q-learning model gives the agent the expected long-term optimal reward, once the agent sees some faults in S_2, it will learn that if it uses one copy, the control inputs would be wrong, and the physical states would enter into S_3 with some probability. This would make the agent more cautious about using just one copy while it is inside S_2.

The design of a more advanced reward function, the tuning of the hyper-parameters and the choice of the q-states are all worthy of future study.

CHAPTER 7

CONCLUSION

As cyber-physical systems become ever more complicated, a trend has emerged towards running the control tasks on an integrated computation platform rather than on isolated controllers. Traditionally, massive always-on redundancy has been used to ensure reliable controller performance. However, in many, if not most, instances, the controlled plant is so deep within its allowed state space that occasional controller errors do not cause controller plant failure. This leads to an adaptive approach to fault-tolerance. Such an approach significantly reduces the computational burden of the controller; this reduced burden leads to lower controller operating temperatures which prolongs mean processor lifetime.

This dissertation describes a generalized software simulation framework, AdaFT, for an adaptive fault-tolerance approach in CPSs. AdaFT first partitions the state space of the controlled plant based on how much fault-tolerance is required, with possibly multiple versions of each control task. Then a machine-learning based approach is used to generate a compact memory look-up table indicating the required level of fault-tolerance. When provided with information about power consumption, the framework carries out a thermal analysis of the cyber elements and uses that information to estimate reliability. Possible extensions to AdaFT include a range of control systems being evaluated (i.e., effective handling of nonlinear and large systems); hierarchical management of distributed applications with multiple, interacting, centers of control; design space exploration for assistance in designing the controller; automatic load tuning/balancing; and hardware provisioning according to a user-defined expected system reliability or lifetime.

This framework can be used in every stage of the design process. It can be used to determine the impact of control task dispatch frequencies on system performance and reliability. Given processor failure rates, its calculation of thermally accelerated aging can determine hardware provisioning for specified operational lifetime. Its analysis of controlled plant dynamics can be used to set the required level of fault-tolerance for safe plant functioning. At a time when CPS complexity and demands on reliability are both increasing, AdaFT facilitates an approach to adaptive fault-tolerance that allows for economical and safe management of resources in cyber-physical systems.

APPENDIX

ADAFT EXAMPLE PROGRAMS

In this section we provide some examples of programs using AdaFT. These code snippets are by no means complete; they only show how users can define their own fault-tolerant CPSs using AdaFT's API. From these templates, it is standard to define a CPS using AdaFT API.

Case 1: Inverted Pendulum Code Snippet. Main Program:

```
### Inverted Pendulum ###
h = 0.001
kf_period = 0.009
kf_deadline = kf_period
kf_wcet = 0.001
kf_power = 6.5

lqr_pen_period = 0.02
lqr_pen_deadline = lqr_pen_period
lqr_pen_wcet = 0.01
lqr_pen_power = 3.5

actuator_noise = {'lqr' : 3}

cart_pos_noise = 0.001
cart_vel_noise = 0.001
angle_noise = 0.01
rate_noise = 0.005
cart_pos_noise1 = 0.001
cart_vel_noise1 = 0.001
angle_noise1 = 0.002
rate_noise1 = 0.1

A = np.array([[0., 1.0, 0., 0.],
              [0., -0.18182, 2.6727, 0.],
              [0., 0., 0., 1.],
              [0., -0.45455, 31.1820, 0.]])
```

```
B = np.array([[0.0],
              [1.8182],
              [0.],
              [4.5455]])

C = np.array([[1., 0., 0., 0.],
              [0., 1., 0., 0.],
              [0., 0., 1., 0.],
              [0., 0., 0., 1.]])

K = np.array([[-61.9930, -33.5054, 95.0600, 18.8300]])
P = np.array([[1000., 0., 0., 0.],
              [0., 1000., 0., 0.],
              [0., 0., 1000., 0.],
              [0., 0., 0., 1000.]])


R0 = np.array([[cart_pos_noise, 0., 0., 0.],
              [0., cart_vel_noise, 0., 0.],
              [0., 0., angle_noise, 0.],
              [0., 0., 0., rate_noise]])
R1 = np.array([[cart_pos_noise1, 0., 0., 0.],
              [0., cart_vel_noise1, 0., 0.],
              [0., 0., angle_noise1, 0.],
              [0., 0., 0., rate_noise1]])

Rs = {'sensor0' : R0, 'sensor1' : R1}

F = 1 + kf_period * (A - np.dot(B, K))



sss = np.zeros((4, 1))
c_sss = np.zeros((4, 1))


cart_pos = [p for p in np.arange(0, 0.5, 0.5)]
cart_vel = [p for p in np.arange(0, 1, 1)]
angles = [p for p in np.arange(0.4, 0.5, 0.1)]
rate = [p for p in np.arange(0.5, 1.5, 1)]


x0 = np.array([[p],
                                  [v],
                                  [a],
                                  [r]])
```

```
init_x = copy.deepcopy(x0)
xs = x0

xtract = np.array([[0.0],
                                        [0.0],
                                        [0.4],
                                        [0.5]])


actuator = Actuator(actuator_noise)

pen = InvPenDynamics(x0, A, B)

sensor = InvPenSensor([cart_pos_noise, cart_vel_noise, angle_noise, rate_nois
sensor1 = InvPenSensor([cart_pos_noise1, cart_vel_noise1, angle_noise1, rate_
sensors = [sensor, sensor1]

kf = ct.makeLinearKF(A, B, C, P, F, x0[:, -1])
localizer = KalmanPredict('filter', kf_period, kf_deadline, kf_wcet, kf_power
lqr = LQRInvPen('lqr', lqr_pen_period, lqr_pen_deadline, lqr_pen_wcet, lqr_pe
kalman_lqr = KalmanLQR('kalman_lqr', lqr_pen_period, lqr_pen_deadline, lqr_pe

queue = [localizer, lqr]
queue2 = [kalman_lqr]
rtos = RTOS()
for task in queue:
        rtos.create_task(task)
fail_rate = 3.1706e-09
reliability_model = TAAF(fail_rate)
processor = Processor(rtos, reliability_model)
processor2 = copy.deepcopy(processor)
processor3 = copy.deepcopy(processor)

cyber = CyberSystem([processor, processor2, processor3])

end = 4

cps = InvPenCPS(pen, cyber, sensors, actuator, end=end)

cps.run()

print("Total number of tasks released: ", cps.cyber_sys.processors[0].rtos.n)
print("Missed tasks: ", cps.cyber_sys.processors[0].rtos.missed_task)

xtract = cps.xtract
xs = cps.xs
taaf = cps.taaf[cps.taaf != 0]
```

```
temp = cps.temp[cps.temp != 0]

plt.figure()
plt.hold(True)
plt.grid(True)
plt.plot(xtract[2, :], c = 'b', label='Kalman Filter', linewidth=2)
plt.plot(xs[2, :], c= 'r', label='True value', linewidth = 1)
plt.xlabel('Time (ms)', fontsize=18)
plt.ylabel('Angle (rad)', fontsize=18)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.legend(loc=4)

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
ax1.plot(taaf)
ax1.set_title('taaf')
ax2.plot(temp)

plt.show()
```

Inverted Pendulum CPS Definition:user needs to implement the stop condition in *should-Stop* method, and the *run* method to customize the running procedure of the CPS. The parent class *CyberPhysicalSystem* already has a *run* method implemented with the minimum update steps for necessary simulations. Users can override it by introducing more customized procedures, such as logging the state trajectories.

```
class InvPenCPS(CyberPhysicalSystem):
    def __init__(self, physical_sys, cyber_sys, sensors, actuator, h = 0.001,
        super().__init__(physical_sys, cyber_sys, sensors, actuator, h)
        self.end = end
        self.taaf = np.zeros([(end + 2 * h) / h, 1])
        self.taaf[0] = 1
        self.temp = np.zeros([(end + 2 * h) / h, 1])
        self.xs = self.physical_sys.x

        self.xtract = self.physical_sys.x



    def should_stop(self):
        return (self.clock >= self.end) or (not self.physical_sys.is_safe())


    def run(self):
        j = 0
```

```
        while not self.should_stop():
            self.step_update()
            self.clock += self.h
            # print(self.clock)
            self.taaf[j] = self.cyber_sys.processors[0].reliability_model.taa
            self.temp[j] = self.cyber_sys.processors[0].reliability_model.abs
            j += 1
            self.xs = np.append(self.xs, self.physical_sys.x, axis=1)
            x_predict = np.reshape(self.cyber_sys.processors[0].rtos.task_out
            self.xtract = np.append(self.xtract, x_predict, axis=1)
```

Inverted Pendulum Physical System: User needs to implement the *update* method, to tell AdaFT how the physical dynamics would update according to its current states and the control laws.

```
from Physical.PhysicalSystem import PhysicalSystem
import numpy as np


class InvPenDynamics(PhysicalSystem):
    def __init__(self, x0, A, B):
        super().__init__(x0)
        self.A = A
        self.B = B

    def update(self, h, clock, actuator_commands):
        self.u = actuator_commands['lqr']
        self.u = np.reshape(self.u, (-1, 1))
        x_dot = np.dot(self.A, self.x) + np.dot(self.B, self.u)
        self.x += h * x_dot

    def is_safe(self):
        return -0.5 <= self.x[2] <= 0.5
```

Inverted Pendulum Sensor Model: User can provide different noise scales for different sensors, by overriding the *sense* method.

```
class InvPenSensor(Sensor):
    def sense(self, x):
        if len(x) != 4:
            raise MyException('The Inverted Pendulum should have 4 physical s

        return np.array([[x[0][0] + randn() * self.noise_scale[0]],
                         [x[1][0] + randn() * self.noise_scale[1]],
                         [x[2][0] + randn() * self.noise_scale[2]],
```

```
                                [x[3][0] + randn() * self.noise_scale[3]]])
```

Inverted Pendulum Tasks, LQR and Kalman Filter: User needs to implement the *density* method, which yields the actual execution time for each iteration of a task. User also needs to implement the *run* method, which is the essential algorithm.

```
class LQRInvPen(TaskModel):
    def __init__(self, name, period, deadline, wcet, power, K):
        super().__init__(name, period, deadline, wcet, power)
        self.K = K
        self.output = np.array([[0]])

    def density(self):
        return self.wcet

    def run(self, inputs):
        self.output = -np.dot(self.K, inputs)


class KalmanPredict(TaskModel):
    """
    Localization task
    """
    def __init__(self, name, period, deadline, wcet, power, kf, Rs):
        """
        Kalman Filter to predict linear systems
        :param kf: kalman filter object
        :param Rs: a dictionary of sensor noise matrix, R
        """
        super().__init__(name, period, deadline, wcet, power)
        self.kf = kf
        self.Rs = Rs
        self.output = self.kf.x

    def density(self):
        return self.wcet

    def run(self, inputs):
        """
        :param inputs: {'lqr':....., 'sensor': {dict of all sensor readings}}
        """
        threshold = 75
        inputs = inputs['sensor']
        self.kf.predict()
        predicted = copy.deepcopy(self.kf.x)
        for sensor_name, z in inputs.items():
            errors = abs(predicted - z)
```

```
            if errors[2] < threshold:
                self.kf.update(z, self.Rs[sensor_name])
            # self.kf.update(z, self.Rs[sensor_name])
        self.output = self.kf.x


class KalmanLQR(TaskModel):
    """
    Localization plut control task
    """
    def __init__(self, name, period, deadline, wcet, power, kf, Rs, K):
        """
        Kalman Filter to predict linear systems
        :param kf: kalman filter object
        :param Rs: a dictionary of sensor noise matrix, R
        """
        super().__init__(name, period, deadline, wcet, power)
        self.kf = kf
        self.Rs = Rs
        self.K = K
        self.output = -np.dot(self.K, self.kf.x)

    def density(self):
        return self.wcet

    def run(self, inputs):
        """
        :param inputs: dictionary of all sensor measurements
        """
        self.kf.predict()
        for sensor_name, z in inputs.items():
            self.kf.update(z, self.Rs[sensor_name])

        self.output = -np.dot(self.K, self.kf.x)
```

Case 2: ABS Main Program:

```
######### Car ABS  #########

### Define parameters
h = 0.001
x0 = np.array([[30.], [30.], [0.]])
xs = x0
mass_quater_car = 250
mass_effective_wheel = 20
road_friction_coeff = 1.

actuator_noise = {'abs' : 0}
```

```
sensor_vehicle_speed_noise = 0.5 # don't use a very small value, which will l
sensor_wheel_speed_noise = 0.5
sensor_pos_noise = 0.5
all_sensor_noise_levels = {'sensor0' : np.array([sensor_vehicle_speed_noise,

period = 0.02
deadline = period
wcet = 0.01
power = 3.5

period_pf = 0.01
deadline_pf = period_pf
wcet_pf = 0.001
power_pf = 6.5
N = 1000


### Create and Initialize Objects
car = VehicleABSDynamics(x0, road_friction_coeff=road_friction_coeff)
car_sensor = ABSSensor([sensor_vehicle_speed_noise, sensor_wheel_speed_noise,
car_actuator = Actuator(actuator_noise)
abs = ABS('abs', period, deadline, wcet, power)

range = np.array([[0., 80.], [0., 80.], [0., 80.]])
std = np.array([1, 1, 1])
pf = ABSParticleFilter(road_friction_coeff, mass_quater_car, mass_effective_w
                       std, N=N, init_state_guess= x0)
localizer = ABSPF('filter', period_pf, deadline_pf, wcet_pf, power_pf, pf)

xtract = x0

queue = [abs, localizer]
rtos = RTOS()
for task in queue:
    rtos.create_task(task)
fail_rate = 3.1706e-09
reliability_model = TAAF(fail_rate)
processor = Processor(rtos, reliability_model)
processor2 = copy.deepcopy(processor)
processor3 = copy.deepcopy(processor)

cyber = CyberSystem([processor, processor2, processor3])

cps = ABSCPS(car, cyber, [car_sensor], car_actuator, end=10)
cps.run()

taaf = cps.taaf[cps.taaf != 0]
```

```
temp = cps.temp[cps.temp != 0]
xtract = cps.xtract
xs = cps.xs

print("Total number of tasks released: ", processor.rtos.n)
print("Missed tasks: ", processor.rtos.missed_task)

### Plot Results
plt.figure()
plt.hold(True)
plt.grid(True)
plt.plot(xtract[0, :], c = 'c', label='Particle Filter Vehicle Speed', linewi
plt.plot(xtract[1, :], c = 'g', label='Particle Filter Wheel Speed', linewidt
plt.plot(xs[0, :], c = 'b', label='Vehicle Speed', linewidth=1)
plt.plot(xs[1, :], c= 'r', label='Wheel Speed', linewidth = 1)
plt.xlabel('Time (ms)', fontsize=18)
plt.ylabel('Speed m/s', fontsize=18)
plt.xticks(fontsize = 15)
plt.yticks(fontsize = 15)
plt.legend(loc=1, fontsize = 15)

plt.figure()
plt.hold(True)
plt.grid(True)
plt.plot(xs[2, :], c = 'b', label='distance', linewidth=2)
plt.xlabel('Time (ms)')
plt.ylabel('Stop Dis (m)')
plt.legend(loc=4)


f, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
ax1.plot(taaf)
ax1.set_title('taaf')
ax2.plot(temp)
ax2.set_title('mttf in years')

plt.show()
```

ABS CPS Definition:

```
class ABSCPS(CyberPhysicalSystem):
    def __init__(self, physical_sys, cyber_sys, sensors, actuator, h = 0.001,
        super().__init__(physical_sys, cyber_sys, sensors, actuator, h)
        self.end = end
        self.taaf = np.zeros([(end + 2 * h) / h, 1])
        self.taaf[0] = 1
        self.temp = np.zeros([(end + 2 * h) / h, 1])
```

```
        self.xs = self.physical_sys.x

        self.xtract = self.physical_sys.x



    def should_stop(self):
        return (self.clock >= self.end) or (self.physical_sys.x[0] <= 1)


    def run(self):
        j = 0
        while not self.should_stop():
            self.step_update()
            self.clock += self.h

            self.taaf[j] = self.cyber_sys.processors[0].reliability_model.taa
            self.temp[j] = self.cyber_sys.processors[0].reliability_model.abs
            j += 1
            self.xs = np.append(self.xs, self.physical_sys.x, axis=1)
            x_predict = np.reshape(self.cyber_sys.processors[0].rtos.task_out
            self.xtract = np.append(self.xtract, x_predict, axis=1)
```

ABS Car Physical Model:

```
class VehicleABSDynamics(PhysicalSystem):
    def __init__(self, x0, mass_quater_car = 250, mass_effective_wheel = 20,
        """
        :param x0: [vehicle speed, wheel speed, vehicle position]
        :param mass_quater_car: kg
        :param mass_effective_wheel: kg
        :param road_friction_coeff:
        """
        super().__init__(x0)
        self.mass_quater_car = mass_quater_car
        self.mass_effective_wheel = mass_effective_wheel
        self.road_friction_coeff = road_friction_coeff

    def update(self, h, clock, actuator_commands):
        actuator_commands = actuator_commands['abs']
        g = 9.81
        v = self.x[0][0]
        w = self.x[1][0]
        x = self.x[2][0]
        slip = self._slip_ratio(v, w)
        friction_force = self.road_friction_coeff * self._mu(slip) * self.mas
        v = v - h * friction_force / self.mass_quater_car
        x = x + h * v
```

```
        w = w + h * (friction_force / self.mass_effective_wheel - actuator_co
        w = max(0., w)

        self.x = np.array([[v], [w], [x]])

    def _slip_ratio(self, v, w):
        return max(0., 1. - float(w) / float(v))

    def _mu(self, slip):
        return -1.1 * np.exp(-20 * slip) + 1.1 - 0.4 * slip

    def is_safe(self):
        v = self.x[0][0]
        w = self.x[1][0]
        slip = self._slip_ratio(v, w)
        return slip <= 0.25 and slip >= 0.05
```

ABS Sensor:

```
class ABSSensor(Sensor):
    def sense(self, x):
        """
        :param x: [vehicle speed, wheel speed, vehicle position]
        :return: sensor measurements of vehicle speed and wheel speed
        """
        if len(x) != 3:
            raise MyException('The ABS should have 3 physical state variables

        return np.array([[x[0][0] + randn() * self.noise_scale[0]],
                         [x[1][0] + randn() * self.noise_scale[1]],
                         [x[2][0] + randn() * self.noise_scale[2]]])
```

ABS Tasks:

```
class ABS(TaskModel):
    def __init__(self, name, period, deadline, wcet, power, hydraulic_speed =
        super().__init__(name, period, deadline, wcet, power)
        self.hydraulic_speed = hydraulic_speed
        self.upper_bound = upper_bound
        self.lower_bound = lower_bound
        self.output = max(self.lower_bound, min(self.upper_bound, self.output

    def density(self):
        return self.wcet

    def run(self, inputs):
        optimal_slip = 0.2
        v = inputs[0][0]
```

```
        w = inputs[1][0]
        slip = self._slip_ratio(v, w)

        if slip > optimal_slip:
            brake = -1
        else:
            brake = 1

        self.output = self.output + self.period * brake * self.hydraulic_spee
        self.output = max(self.lower_bound, min(self.upper_bound, self.output


    def _slip_ratio(self, v, w):
        return max(0., 1. - float(w) / float(v))


class ABSPF(TaskModel):
    """Localization task"""
    def __init__(self, name, period, deadline, wcet, power, pf):
        super().__init__(name, period, deadline, wcet, power)
        self.pf = pf
        self.output = self.pf.estimate()

    def density(self):
        return self.wcet

    def run(self, inputs):
        """
        :param inputs: dictionary of all sensor measurements, and actuator co
               e.g., {'abs' : 100,
                      'sensor' : {'sensor1' : np.array[30., 30., 10.]
                                  'sensor2' : np.array[32., 31., 10.]
                                  }
                     }
        """
        self.pf.predict(inputs['abs'])

        self.pf.update(inputs['sensor'])
        if self.pf._neff() < self.pf.N / 2:
            self.pf.resample()

        self.output = self.pf.estimate()
```

Case 3: Humanoid Robot Main Program:

```
A = np.array([[   0.   ,    0.   ,    0.   ,    1.   ,    0.   ,
0.   ],
```

```
               [   0.   ,    0.   ,    0.   ,    0.   ,    1.   ,
0.   ],
               [   0.   ,    0.   ,    0.   ,    0.   ,    0.   ,
1.   ],
               [  18.337,  -75.864,    6.395,    0.   ,    0.   ,
0.   ],
               [ -22.175,  230.549,  -49.01 ,    0.   ,    0.   ,
0.   ],
               [   4.353, -175.393,   95.29 ,    0.   ,    0.   ,
0.   ]])

B = np.array([[ 0.   ,  0.   ,  0.   ],
               [ 0.   ,  0.   ,  0.   ],
               [ 0.   ,  0.   ,  0.   ],
               [ 0.292, -0.785,  0.558],
               [-0.785,  2.457, -2.178],
               [ 0.558, -2.178,  2.601]])

C = eye(6)


K = np.array([[1054.367, 426.901, 153.864, 365.784, 173.577, 67.28 ],
              [707.669, 610.181, 251.283, 263.836, 158.583, 72.18 ],
              [12.129,  43.669, 132.469,  11.613,  16.447,  19.12 ]])

K1 = np.array([[ 929.79 ,  322.082,   73.883,  317.782,  142.912,
49.865],
       [ 521.09 ,  431.75 ,   74.787,  188.568,  105.95 ,   38.039],
       [ 277.   ,  239.326,  202.902,  108.11 ,   70.849,   42.213]])

h = 0.001
kf_period = 0.01
kf_deadline = kf_period
kf_wcet = 0.001
kf_power = 6.5

lqr_period = 0.01
lqr_deadline = lqr_period
lqr_wcet = 0.008
lqr_power = 10

lqr_period1 = 0.01
lqr_deadline1 = lqr_period1
lqr_wcet1 = 0.008
lqr_power1 = 10

lqr_period2 = 0.02
```

```
lqr_deadline2 = lqr_period2
lqr_wcet2 = 0.008
lqr_power2 = 10

actuator_noise = {'lqr' : 0}

theta1_noise = 0.0001
theta2_noise = 0.0001
theta3_noise = 0.0001
rate1_noise = 0.0001
rate2_noise = 0.0001
rate3_noise = 0.0001

theta1_noise1 = 0.0001
theta2_noise1 = 0.0001
theta3_noise1 = 0.0001
rate1_noise1 = 0.0001
rate2_noise1 = 0.0001
rate3_noise1 =0.0001

P = 1000 * eye(6)


R0 = np.array([[theta1_noise, 0., 0., 0., 0., 0.],
               [0., theta2_noise, 0., 0., 0., 0.],
               [0., 0., theta3_noise, 0., 0., 0.],
               [0., 0., 0., rate1_noise,  0., 0.],
               [0., 0., 0., 0., rate2_noise,  0.],
               [0., 0., 0., 0., 0., rate3_noise ]])


R1 = np.array([[theta1_noise1, 0., 0., 0., 0., 0.],
               [0., theta2_noise1, 0., 0., 0., 0.],
               [0., 0., theta3_noise1, 0., 0., 0.],
               [0., 0., 0., rate1_noise1,  0., 0.],
               [0., 0., 0., 0., rate2_noise1,  0.],
               [0., 0., 0., 0., 0., rate3_noise1 ]])

Rs = {'sensor0' : R0, 'sensor1' : R1}

F = 1 + kf_period * (A - np.dot(B, K))
F1 = 1 + kf_period * (A - np.dot(B, K1))


x0 = np.array([[-0.45],
               [0.087],
               [-0.087],
```

```
                [-0.087],
                [-0.087],
                [-0.087]])

init_x = copy.deepcopy(x0)
xs = x0

xtract = copy.deepcopy(x0)


actuator = RobotActuator(actuator_noise)

actuator_multi = RobotActuator_multi_control(actuator_noise)

robot = Robot(x0, A, B)

sensor = RobotSensor([theta1_noise, theta2_noise, theta3_noise, rate1_noise,
sensor1 = RobotSensor([theta1_noise1, theta2_noise1, theta3_noise1, rate1_noi
sensors = [sensor, sensor1]

kf = ct.makeLinearKF(A, B, C, P, F, x0[:, -1], 6, 6)
kf1 = ct.makeLinearKF(A, B, C, P, F1, x0[:, -1], 6, 6)

localizer = KalmanPredict('filter', kf_period, kf_deadline, kf_wcet, kf_power
localizer1 = KalmanPredict('filter', kf_period, kf_deadline, kf_wcet, kf_powe

lqr = LQRRobot('lqr', lqr_period, lqr_deadline, lqr_wcet, lqr_power, K)
lqr1 = LQRRobot('lqr', lqr_period, lqr_deadline, lqr_wcet/2, 6, K1)

u1_1 = LQRRobotU1('u1', lqr_period, lqr_deadline, lqr_wcet, lqr_power, K)
u1_2 = LQRRobotU1('u1', lqr_period, lqr_deadline, lqr_wcet/2, 6, K1)

u2_1 = LQRRobotU2('u2', lqr_period1, lqr_deadline1, lqr_wcet1, lqr_power1, K)
u2_2 = LQRRobotU2('u2', lqr_period1, lqr_deadline1, lqr_wcet1/2, 6, K1)

u3_1 = LQRRobotU3('u3', lqr_period2, lqr_deadline2, lqr_wcet2, lqr_power2, K)
u3_2 = LQRRobotU3('u3', lqr_period2, lqr_deadline2, lqr_wcet2/2, 6, K1)

rtos = RTOS()

fail_rate = 3.1706e-09
reliability_model = TAAF(fail_rate)
processor = Processor(rtos, reliability_model)

processor_list = [copy.deepcopy(processor) for i in range(8)]

task_list = {'u1' : (u1_1, u1_2), 'u2' : (u2_1, u2_2), 'u3' : (u3_1, u3_2), '
```

```
clf = pickle.load(open(os.path.join('./Cyber/', 'subspace_clf_decision_tree.p

clf = {'u1' : clf, 'u2' : clf, 'u3' : clf}

cyber = CyberSystem(processor_list, clf, task_list)

end = 5

cps = RobotCPS(robot, cyber, sensors, actuator_multi, end=end)

cps.run()
```

Humanoid Robot CPS Definition:

```
class RobotCPS(CyberPhysicalSystem):
    def __init__(self, physical_sys, cyber_sys, sensors, actuator, h = 0.001,
        super().__init__(physical_sys, cyber_sys, sensors, actuator, h)
        self.end = end
        self.taaf = np.zeros([(end + 2 * h) / h, 1])
        self.taaf[0] = 1
        self.temp = np.zeros([(end + 2 * h) / h, 1])
        self.xs = np.zeros([6, (end + 2 * h) / h])
        self.xs[:, 0] = self.physical_sys.x[:, 0]
        self.xtract = np.zeros([6, (end + 2 * h) / h])
        self.xtract[:, 0] = self.physical_sys.x[:, 0]
        self.copies = np.zeros([1, (end + 2 * h) / h])
        self.version = np.zeros([1, (end + 2 * h) / h])



    def should_stop(self):
        return (self.clock >= self.end) or (not self.physical_sys.is_safe())


    def run(self):
        j = 0
        while not self.should_stop():
            self.step_update()
            self.clock += self.h
            # print(self.clock)
            self.taaf[j] = self.cyber_sys.processors[0].reliability_model.taa
            self.temp[j] = self.cyber_sys.processors[0].reliability_model.abs
            self.xs[:, j] = self.physical_sys.x[:, 0]
            x_predict = np.reshape(self.cyber_sys.processors[0].rtos.task_out
            self.xtract[:, j] = x_predict[:,0]
            self.copies[:, j] = self.cyber_sys.copies
            self.version[:, j] = self.cyber_sys.version
```

```
            j += 1
```

Humanoid Robot Physical Model:

```
class Robot(PhysicalSystem):
    def __init__(self, x0, A, B):
        super().__init__(x0)
        self.A = A
        self.B = B

    def update(self, h, clock, actuator_commands):
        self.u = actuator_commands['lqr']
        self.u = np.reshape(self.u, (-1, 1))
        x_dot = np.dot(self.A, self.x) + np.dot(self.B, self.u)
        self.x += h * x_dot

    def is_safe(self):
        return True
```

Humanoid Robot Sensor:

```
class RobotSensor(Sensor):
    def sense(self, x):
        if len(x) != 6:
            raise MyException('The Robot should have 6 physical state variabl

        return np.array([[x[0][0] + randn() * self.noise_scale[0]],
                         [x[1][0] + randn() * self.noise_scale[1]],
                         [x[2][0] + randn() * self.noise_scale[2]],
                         [x[3][0] + randn() * self.noise_scale[3]],
                         [x[4][0] + randn() * self.noise_scale[4]],
                         [x[5][0] + randn() * self.noise_scale[5]]])
```

Humanoid Robot Tasks:

```
class LQRRobot(TaskModel):
    def __init__(self, name, period, deadline, wcet, power, K):
        super().__init__(name, period, deadline, wcet, power)
        self.K = K
        self.output = np.array([[0]])

    def density(self):
        return self.wcet

    def run(self, inputs):
        self.output = -np.dot(self.K, inputs)

class LQRRobotU1(TaskModel):
    def __init__(self, name, period, deadline, wcet, power, K):
```

```
        super().__init__(name, period, deadline, wcet, power)
        self.K = K
        self.output = np.array([[0]])

    def density(self):
        return self.wcet

    def run(self, inputs):
        self.output = -np.dot(self.K, inputs)[0]

class LQRRobotU2(TaskModel):
    def __init__(self, name, period, deadline, wcet, power, K):
        super().__init__(name, period, deadline, wcet, power)
        self.K = K
        self.output = np.array([[0]])

    def density(self):
        return self.wcet

    def run(self, inputs):
        self.output = -np.dot(self.K, inputs)[1]

class LQRRobotU3(TaskModel):
    def __init__(self, name, period, deadline, wcet, power, K):
        super().__init__(name, period, deadline, wcet, power)
        self.K = K
        self.output = np.array([[0]])

    def density(self):
        return self.wcet

    def run(self, inputs):
        self.output = -np.dot(self.K, inputs)[2]


def makeLinearKF(A, B, C, P, F, x, R = None, dim_x = 4, dim_z = 4):

    kf = KalmanFilter(dim_x=dim_x, dim_z=dim_z)
    kf.x = x
    kf.P = P
    kf.F = F
    kf.H = C
    kf.R = R

    return kf
```

BIBLIOGRAPHY

[1] Guidelines for the use of c language in critical systems.

[2] Aydin, Hakan, Melhem, Rami, Mossé, Daniel, and Mejía-Alvarez, Pedro. Power-aware scheduling for periodic real-time tasks. *Computers, IEEE Transactions on 53*, 5 (2004), 584–600.

[3] Bak, S., Johnson, T.T., Caccamo, M., and Sha, L. Real-time reachability for verified simplex design. *Real-Time Systems Symposium (RTSS)* (2014).

[4] Bergenheim, C., Shladover, S., and Coelingh, E. Overview of platooning systems. *Proceedings of the 19th ITS World Congress* (2012).

[5] Bhatti, Muhammad Khurram, Belleudy, Cécile, and Auguin, Michel. An inter-task real time dvfs scheme for multiprocessor embedded systems. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on* (2010), IEEE, pp. 136–143.

[6] Bogdan, P., and Marculescu, R. Towards a science of cyber-physical systems design. In *Cyber-Physical Systems (ICCPS), 2011 IEEE/ACM International Conference on* (2011), IEEE, pp. 99–108.

[7] Bogdan, Paul, Dumitraş, Tudor, and Marculescu, Radu. Stochastic communication: A new paradigm for fault-tolerant networks-on-chip. *VLSI design 2007* (2007).

[8] Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning* (2013), pp. 108–122.

[9] Burns, A., and Davis, R.I. Mixed criticality systems – a review. *IEEE Real-Time Systems Symposium (RTSS)* (2015).

[10] Burns, A., and Wellings, A. *Real-Time Systems and Programming Languages*. Academic Press, 2009.

[11] Burns, Alan, and Davis, Robert. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep* (2013).

[12] Chen, Jian-Jia. Expected energy consumption minimization in dvs systems with discrete frequencies. In *Proceedings of the 2008 ACM symposium on Applied computing* (2008), ACM, pp. 1720–1725.

[13] Chen, Jian-Jia, Yang, Chuan-Yue, Kuo, Tei-Wei, and Shih, Chi-Sheng. Energy-efficient real-time task scheduling in multiprocessor dvs systems. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference* (2007), IEEE Computer Society, pp. 342–349.

[14] Clausen, J. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen* (1999), 1–30.

[15] Cooling, J. *Real-time Operating Systems*. Lindentree Associates, 2013.

[16] Currier, P. N. A method for modeling and prediction of ground vehicle dynamics and stability in autonomous systems. *PhD Thesis in Virginia Tech University* (2011).

[17] Dugoff, H., Fancher, P. S., and Segel, L. Tire performance characteristics affecting vehicle response to steering and braking control inputs. *Transportation Research Board* (1969).

[18] Edward A. Lee, Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*, 2 ed. 2015.

[19] Fraga, J., Siqueira, F., and Favarim, F. An adaptive fault-tolerant component model. *IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* (2003).

[20] Goldberg, J., Greenberg, I., and Lawrence, T.F. Adaptive fault tolerance. *IEEE Workshop on Advances in Parallel and Distributed Systems* (1993).

[21] Goyal, A., and Tantawi, A. N. Evaluation of performability for degradable computer systems. *IEEE Transactions on Computers 100* (1987).

[22] Kalyanasundaram, Bala, and Pruhs, Kirk R. Fault-tolerant scheduling. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing* (1994), ACM, pp. 115–124.

[23] Koren, I., and Krishna, C. M. Fault-tolerant systems. *Morgan Kaufmann* (2007).

[24] Krishna, C.M. Ameliorating thermally acclerated aging with state-based application of fault-tolerance in cyber-physical computers. *IEEE Transactions on Reliability 64* (2015).

[25] Krishna, C.M., and Koren, I. Adaptive fault-tolerance for cyber-physical systems. *CPS Workshop* (2013).

[26] Krishna, C.M., and Shin, K.G. Performance measures for control computers. *IEEE Transactions on Automatic Control 32* (1987).

[27] Lee, Edward, et al. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on* (2008), IEEE, pp. 363–369.

[28] Lehoczky, J., Sha, L., and Ding, Y. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.* (1989), IEEE, pp. 166–171.

[29] Li, M., Ramachandran, P., Sahoo, S. Kumar, Adve, S. V, Adve, V. S, and Zhou, Y. Understanding the propagation of hard errors to software and implications for resilient system design. *ACM SIGARCH Computer Architecture News* (2008).

[30] Li, Sheng, Ahn, Jung Ho, Strong, Richard D, Brockman, Jay B, Tullsen, Dean M, and Jouppi, Norman P. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), ACM, pp. 469–480.

[31] Liestman, Arthur L, and Campbell, Roy H. A fault-tolerant scheduling problem. *Software Engineering, IEEE Transactions on*, 11 (1986), 1089–1095.

[32] Liu, X., Wang, Q., Gopalakrishnan, S., He, W., Sha, L., Ding, H., and Lee, K. Ortega: An efficient and flexible online fault tolerance architecture for real-time control systems. *IEEE Transactions on Industrial Informatics* (2008).

[33] Meyer, J. F. Closed-form solutions of performability. *IEEE Transactions on Computers* (1982).

[34] Meyer, J. F., Furchtgott, D. G., and Wu, L. T. Performability evaluation of the sift computer. *IEEE Transactions on Computers* (1980).

[35] Michigan, University. Matlab and simulink tutorial.

[36] Moazzami, R., Lee, J. C, and Hu, C. Temperature acceleration of time-dependent dielectric breakdown. *IEEE Transactions on Electron Devices* (1989).

[37] Moore, J., and Crist, J. Multibody dynamics using python.

[38] Murphy, K. P. Machine learning: a probabilistic perspective. *MIT Press* (2013).

[39] P.A. Laplante, S.J. Ovaska. *Real-Time Systems Design and Analysis, Tools for the Practitioner*, 4 ed. 2011.

[40] Pont, M.J. *The Engineering of Reliable Embedded Systems*. SafeTTy Systems, 2015.

[41] Ptolemaeus, Claudius, Ed. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[42] Rajamani, R. Vehicle dynamics and control. *Springer* (2011).

[43] Rajkumar, Ragunathan Raj, Lee, Insup, Sha, Lui, and Stankovic, John. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference* (2010), ACM, pp. 731–736.

[44] S. Bak, K.K. Chivukula, O. Adekunle M. Sun M. Caccamo L. Sha. The system-level simplex architecture for improved real-time embedded system safety. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2009).

[45] Schoen, J. A model of electromigration failure under pulsed condition. *Journal of Applied Physics* (1980).

[46] Schroder, D. K. Negative bias temperature instability: What do we understand? *Microelectronics Reliability* (2007).

[47] Shin, K.G., Krishna, C.M., and Lee, Y.-H. A unified method for evaluating real-time computers and its application. *IEEE Transactions on Automatic Control* (1985).

[48] Simulation, Mechanical. Carsim.

[49] Skadron, K., Stan, M. R., Sankaranarayanan, K., Huang, W., Velusamy, S., and Tarjan, David. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimization (TACO)* (2004).

[50] Song, J., and Parmer, G. Cmon: a predictable monitoring infrastructure for system-level latent fault detection and recovery. *Real-Time and Embedded Technology and Application Symposium (RTAS)* (2015).

[51] Song, J., Wittrock, J., and Parme, G. Predictable, efficient system-level fault tolerance in cˆ 3. *Real-Time Systems Symposium (RTSS)* (2013).

[52] Stankovic, John A, Spuri, Marco, Ramamritham, Krithi, and Buttazzo, Giorgio C. Introduction. In *Deadline Scheduling for Real-Time Systems*. Springer, 1998, pp. 1–11.

[53] Stevens, Brian L, Lewis, Frank L, and Johnson, Eric N. *Aircraft Control and Simulation: Dynamics, Controls Design, and Autonomous Systems*. John Wiley & Sons, 2015.

[54] Sutton, Richard S, and Barto, Andrew G. *Reinforcement learning: An introduction*. MIT press, 1998.

[55] Swetha, A., V, R. Pillay, and Punnekkat, S. Design, analysis and implementation of improved adaptive fault tolerant model for cruise control multiprocessor system. *International Journal of Computer Applications (IJCA)* (2014).

[56] Thrun, S., Burgard, W., and Fox, D. *Probabilistic robotics*. MIT press, 2005.

[57] Vestal, S. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. *IEEE Real-Time Systems Symposium (RTSS)* (2007).

[58] Vigrass, W. J. Calculation of semiconductor failure rates. *Harris Semiconductor* (2004).

[59] Wittenmark, B. Computer-controlled systems: theory and design. *Courier Dover Publications* (2011).

[60] Zhuo, Jianli, Chakrabarti, Chaitali, and Chang, Naehyuck. Energy management of dvs-dpm enabled embedded systems powered by fuel cell-battery hybrid source. In *Proceedings of the 2007 international symposium on Low power electronics and design* (2007), ACM, pp. 322–327.