TOWARDS REAL TIME OPTIMAL AUTO-TUNING OF PID CONTROLLERS

A Dissertation

by

AARON JAMISON HILL

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2013

Major Subject: Electrical Engineering

ABSTRACT

The Proportional-Integral-Derivative (PID) controller has been widely used by the process control industry for many years. Design methods for PID Controllers are mature and have been heavily researched and evaluated. For most of its modern history the Ziegler-Nichols methods have been used for tuning PID controllers into desired operating conditions. Recently, automatic tuning methods have been formulated and used to generate stable PID controlled systems. These methods have also been implemented on real time systems. However, the use of optimal methods for auto tuning PID controllers on real time systems has not seen much discussion. In this thesis we explore the applicability of optimal PID design methods from Datta, Ho, and Bhattacharrya, to real time system control. The design method is based on a complete characterization of the set of stabilizing PID parameters for various plant models and a subsequent search over the stabilizing set for the optimal controller. A full implementation of the algorithms are completed on an embedded system with DSP hardware. These implementations are then tested against a large number of examples to determine both accuracy and applicability to real time systems.

The major design constraint for application of these algorithms to real time systems is computation time. The faster the optimal result can be computed the more applicable the algorithm is to a real time environment. In order to bring each of these algorithms into a real time system, fast search algorithms were developed to quickly compute the optimal result for the given performance criterion. Three different search methods were developed, compared and analyzed. The first method is a brute force search used as a basis to compare the two additional fast search methods. The two faster search methods prove to be vastly superior in determining the optimal result with the same level of accuracy as brute force search, but in a greatly reduced time. These search methods achieve their superior speeds by reducing the search space without sacrificing accuracy of the results. With these two fast search methods applied to the complete characterization of stabilizing PID controllers, application to real time systems is achieved and demonstrated through examples of various performance criteria.

DEDICATION

To my loving wife, Olivia, and to my parents, brother, sisters and closest friends.

ACKNOWLEDGEMENTS

There have been a great number of people in my life over the length of my PhD studies at Texas A&M who have been instrumental in the completion of my time here. I want to take some time to address some of these people.

I would like to thank my advisor, Dr. Datta, who saw in me a capable and qualified student, and provided me the opportunity to complete. Additionally, my committee members, Dr. Bhattacharyya and Dr. Swaroop for being a part of my time here, and Dr. Karen Buttler-Pury for providing me wise advice and council in hard times. And for Dr. Henry Pfister for being a part of my defense as a substitute committee member.

I want to thank my wife Olivia. The sacrifice taken in the first 2 years of our marriage for me to complete this work has been tremendous. She has stood by my side in support, lifting me up and encouraging me to keep going. She is a true blessing in my life.

To my parents, Kevin and Lu Ann, who supported me through undergraduate education and who afforded me the opportunity to support myself through graduate school by extending me employment in their business. And for their loving and kind support through all the years.

To my siblings, Allyson, Matthew, and Madison, whom have been there for me always. And to my father, my uncle Doug, Dr. Mike Buckley, Dr. Aitzaz Ahmad and Carlos, for their support and encouragement to continue my work.

NOMENCLATURE

$\deg(P(s))$	The degree of the polynomial $P(s)$
DSP	Digital Signal Processor
card(X)	The cardinality of the set X
k_p	The proportional gain value of a PID controller
k_i	The integral gain value of a PID controller
k_d	The derivative gain value of a PID controller
K_p	The interval of k_p values
K_i	The interval of k_i values
K_d	The interval of k_d values
$\|x\|$	The $\mathcal{L}_2$ vector norm of x
$\|A\|$	The matrix norm of A
$\langle F, G\rangle$	The inner product of F and G
$OLHP$	The open left half plane
$ORHP$	The open right half plane
LTI	Linear Time Invariant

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Error feedback control has traditionally been a cornerstone of control theory. The output being controlled is measured and compared to a target value. This comparison generates an error signal which is fed into the controller where it is manipulated. The manipulation of the error signal to generate the control input is done in a manner that stabilizes the system's output, possibly around a known operating condition. One of the most widely used error feedback control structures in industry applications is known as the Proportional, Integral, and Derivative (PID) Controller. The PID controller is popular because it provides good performance under wide operating conditions and is functionally simple to implement [1]. The simplicity arises from the need to only specify three gain parameters to define the PID controller; the proportional, integral, and derivative gains, or k_p, k_i, and k_d, values. A block diagram of this controller is presented in Figure 1.1.

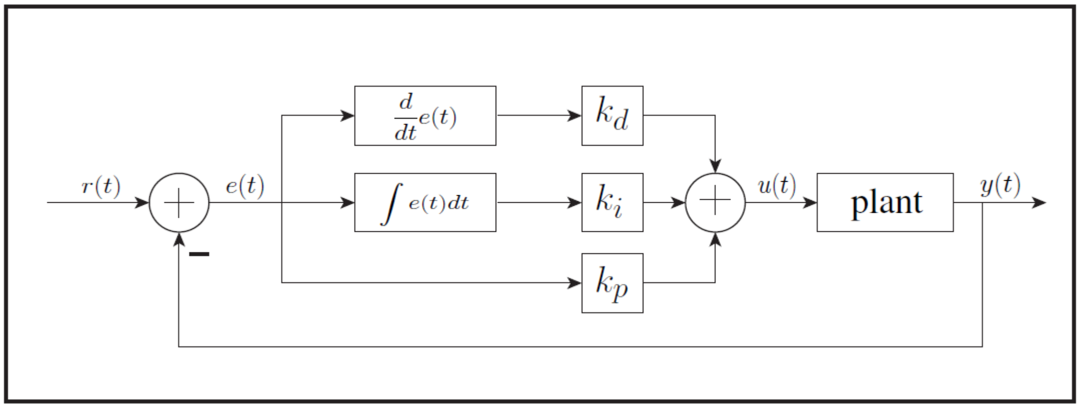

Figure 1.1: Block diagram of a PID controller in a system diagram.

The proportional gain, k_p, creates a constant gain stabilization factor. The integral gain, k_i, provides asymptotic step tracking and drives the error signal to zero. The derivative gain, k_d, allows for transient response shaping. Formally, in the time-domain, the PID controller is defined as

$$u(t) = k_p e(t) + k_i \int e(t)dt + k_d \frac{d}{dt} e(t)$$

or in the laplace domain as

$$U(s) = \left[k_p + \frac{k_i}{s} + k_d s \right] E(s) \ .$$

The integral term is by far the most important term used in PID design because it drives the error to zero, regulating the output of the system around a specified operating condition. The derivative term is often avoided because it will naturally amplify high frequency noise.

The use of PID controllers dates back to early work in governor design in the 1890s [2, 3]. Possibly the first published theoretical analysis of a PID controller was done by Nicolas Minorsky [4]. This research was based on designs for the US Navy to generate automatic steering mechanisms for various naval vessels. In the 1940s PID controllers began to receive much attention in the research communities led by Ziegler and Nichols [5]. A portion of their work meticulously examined many of the characteristics of PID controllers and sought to provide standard names and unit measurements for them. The first PID controllers were purely mechanical devices. With the advent of analog circuits, electro-mechanical PID controllers surfaced. In current times, most PID controllers are digital and implemented on micro controllers, Field Programable Gate Arrays (FPGAs) or Programable Logic Controllers (PLC).

Co-evolving along side PID controllers were various tuning methods used to obtain desired response characteristics with as little trial-and-error as possible. A comprehensive treatment of tuning methods can be found in [6]. Some of the more popular off-line methods used are the Ziegler-Nichols Step and Frequency response method, The Cohen-Coon Method, and the Internal Model Control method. Table 1.1 shows how adjustment of the three PID parameters affects the system's response characteristics [7]. As research efforts improved and technology became more advanced, the topic of auto-tuning of PID controllers was explored. Auto-tuning is concerned with algorithmic approaches that automatically determine the k_p, k_i, and k_d parameters needed to obtain some desired system response characteristics or stability. These methods can be performed either off-line or online.

Table 1.1: Qualitative effect of each gain on the overall system characteristics.

GAINS	RISE TIME	OVERSHOOT	SETTLING TIME	S-S ERROR
k_p	Decrease	Increase	Small Change	Decrease
k_i	Decrease	Increase	Increase	Eliminate
k_d	Small Change	Decrease	Decrease	No Change

The literature is rich and diverse in the area of auto-tuning methods for PID controllers. The Relay Feedback Approach [8, 9] is one of those methods. This approach is a two step process. First it brings the system to an oscillatory state, and then the controller parameters are determined based on the period and amplitude of the oscillation. Genetic algorithms [10, 11, 12, 13] have been another popular approach in auto-tuning methods, as well as neural network based approaches [14, 15, 16]. Still, other methods [17, 18] exist but none guarantee optimality based on a given performance criterion. In the context of optimal auto-tuning techniques, [19] presents a method for minimization of the time weighted integral performance criteria of first order plants with dead time. In [20] a method for auto-tuning is performed on-line in real time with no knowledge of the plant using a search procedure that has a linear quadratic performance index. Keel et al., introduce a nonparametric, and model independent characterization of arbitrary order stabilizing controllers in [21, 22] which could be used for an on-line auto tuning method for PID controllers.

As mentioned above, nearly all PID controllers are implemented in digital hardware on embedded systems such as, FPGAs, PLC, microcontrollers, and even DSPs. The Digital Signal Processor (DSP) offers several advantages over the General Purpose Processors (GPP) for certain types of problems which can be formulated mathematically. DSPs are specialized microprocessors optimized for digital signal processing algorithms: audio and video compression, speech recognition, digital image processing, and digital communication to name a few. DSPs offer special arithmetic operations such as fast multiply-accumulate operations. Many core digital signal processing algorithms like FIR filters and Fast Fourier Transforms rely heavily on multiply-accumulates performance. A GPP system allows for the versatility of performing nearly any task the programmer requires. Most GPP systems can implement digital signal processing algorithms successfully, but the difference in speed, power, and parallelization is noticeable. While a GPP may be more versatile in its programmability, it suffers from slowdowns in handling the extra overhead of the instruction set that exists to create such a rich programming basis. The DSP does not suffer from this slowdown and it is known that DSPs operate much faster than GPPs [23] in the realm of problems that DSPs are applied to. In addition DSPs offer great power savings in the computation per watt metric and many of the problems they are tasked with lend well to parallelism.

Since DSPs have a computational advantage over GPP systems, we want to explore their use in implementing optimal real time auto tuning methods for PID controllers. This would have a great application in systems where the plant parameters may change over time. Being able to exploit the simplicity of a PID controller for controlling a dynamic plant while guaranteeing optimal performance would be very useful. As a basis for our tuning methods we use [24] where a detailed theory for the complete characterization of stabilizing PID parameters is given for any plant order. Such characterization is provided for linear time-invariant plants, interval plants, first-order systems with time delay, discrete time plants and constant gain stabilization with a desired damping. In each case the complete set of stabilizing P, PI, and/or PID controllers are generated. Based on these results, a search method for optimal controller design based on different performance criteria can be developed.

We fully implement all the algorithms of [24] onto a Texas Instruments (TI) DSP evaluation board. The DSP hardware chosen is a TMDSEVM6678LE Evaluation Module which makes available the TMS320C6678 embedded DSP processor from Texas Instruments. The TMS320C6678 contains eight processing cores at 1.0 to 1.25 GHz clock frequencies offering 320 GMAC/160 GFLOP[1] at 1.25GHz. Additionally each core has 32KB L1P and L1D memory, 512KB of L2 memory, and 4MB shared L2 memory for all cores. The evaluation board comes equipped with 512MB of DDR3-1600 RAM. Our design environment was a Microsoft Windows 7 PC with an AMD Phenom 9500 Quad-Core processor at 2.20 GHz with 8 GB of DDR3 RAM hosting Ti's Code Composer Studio version 5.3 for programming/debuging the DSP evaluation board via the onboard XDS560v2 emulator. The DSP was coded mainly in Ti DSP C++ with a few submodules codded in Ti DSP C. This DSP system forms the basis for our real time optimal auto tuning methods for PID controllers.

[1]GMAC is a Giga Multiply-Accumulate Operations and a GFLOP is a Giga Floating Point Operation

2. EMBEDDED SYSTEMS IMPLEMENTATION OF THE STABILIZATION OF LINEAR TIME-INVARIANT PLANTS USING P, PI, AND PID CONTROLLERS

When formulating the analytical algorithms of [24] into a computational algorithm to be implemented onto an embedded system, we leave out much of the technical theory that is behind the formulation. We strive to create a method that includes only the bare necessities of the mechanics of computation carried out in order to produce the end result. In this chapter we build the core computational algorithm for an embedded system which will serve as the foundation for all future discussions. In all cases we consider the following plant model

$$G(s) = \frac{N(s)}{D(s)}$$

$$\begin{aligned} N(s) &= a_m s^m + a_{m-1} s^{m-1} + \cdots + a_1 s + a_0 \\ D(s) &= b_n s^n + b_{n-1} s^{n-1} + \cdots + b_1 s + b_0 \end{aligned}$$

where $n \geq m$. We will first define some of the elementary details from [24] to lay a foundation for further discussion. The numerator and denominator polynomials, $N(s)$ and $D(s)$, respectively, have even and odd decompositions defined as

$$\begin{aligned} N(s) &= N_e(s^2) + sN_o(s^2) \\ D(s) &= D_e(s^2) + sD_o(s^2) \ . \end{aligned}$$

and we also define the term $N^*(s) = N(-s)$. Let $\delta(s, \ldots)$ be the closed loop characteristic polynomial of the system illustrated in Figure 1.1. For P, PI, and PID controllers this polynomial is

$$\begin{aligned}
\delta(s, k_p) &= D(s) + k_p N(s) \\
\delta(s, k_p, k_i) &= sD(s) + (k_i + k_p s)N(s) \\
\delta(s, k_p, k_i, k_d) &= sD(s) + (k_i + k_p s + k_d s^2)N(s) \ ,
\end{aligned}$$

respectively. Now, [24] defines $n = deg(\delta(s, \ldots))$ and $m = deg(N(s))$. An important part of the theory in [24] is derived from the term $\delta(s, \ldots)N^*(s)$ which aids in determining system stability and is mainly utilized in the frequency-domain as $\delta(j\omega, \ldots)N^*(j\omega)$, after substituting $s = j\omega$. In the case of constant gain stabilization this generates the expression

$$\delta(j\omega, k_p)N^*(j\omega) = [p_1(\omega) + k_p p_2(\omega)] + jq(\omega)$$

where

$$\begin{aligned}
p_1(\omega) &= [D_e(-\omega^2)N_e(-\omega^2) + \omega^2 D_o(-\omega^2)N_o(-\omega^2)] \\
p_2(\omega) &= [N_e(-\omega^2)N_e(-\omega^2) + \omega^2 N_o(-\omega^2)N_o(-\omega^2)] \\
q(\omega) &= \omega[N_e(-\omega^2)D_o(-\omega^2) - D_e(-\omega^2)N_o(-\omega^2)] \ .
\end{aligned}$$

But for PI and PID we have

$$\begin{aligned}
\delta(j\omega, k_p, k_i)N^*(j\omega) &= [p_1(\omega) + k_i p_2(\omega)] + j[q_1(\omega) + k_p q_2(\omega)] \\
\delta(j\omega, k_p, k_i, k_d)N^*(j\omega) &= \left[p_1(\omega) + (k_i - k_d\omega^2)p_2(\omega)\right] + j[q_1(\omega) + k_p q_2(\omega)] \ ,
\end{aligned}$$

respectively, where

$$\begin{aligned}
p_1(\omega) &= -\omega^2[N_e(-\omega^2)D_o(-\omega^2) - D_e(-\omega^2)N_o(-\omega^2)] \\
p_2(\omega) &= [N_e(-\omega^2)N_e(-\omega^2) + \omega^2 N_o(-\omega^2)N_o(-\omega^2)] \\
q_1(\omega) &= \omega[D_e(-\omega^2)N_e(-\omega^2) + \omega^2 D_o(-\omega^2)N_o(-\omega^2)] \\
q_2(\omega) &= \omega[N_e(-\omega^2)D_e(-\omega^2) + \omega^2 N_o(-\omega^2)N_o(-\omega^2)] \ .
\end{aligned}$$

Further, the normalized polynomials are defined as

$$\begin{aligned}
p_f(\omega) &= \frac{p(\omega)}{(1+\omega^2)^{\frac{m+n}{2}}} \\
q_f(\omega) &= \frac{q(\omega)}{(1+\omega^2)^{\frac{m+n}{2}}} \ .
\end{aligned}$$

The formal statement of the main result in [24] for the constant gain stabilization, PI, and PID algorithms involve certain *strings* of the real numbers 0, 1, and -1. For clarity of presentation we need to define these strings precisely.

Definition 1. *Let the integers m, n and the function $q_f(\omega)$ be as already defined. Let $0 = \omega_0 < \omega_1 < \ldots < \omega_{l-1}$ be the real, non-negatibve, distinct finite zeros of $q_f(\omega)$ with odd multiplicities. Define a sequence of numbers $i_0, i_1, \ldots, i_l$ as follows:*
(i) If $N^*(j\omega_t) = 0$ for some $t = 1, 2, \ldots, l-1$, then define $i_t = 0$;
(ii) If $N^*(s)$ has a zero of multiplicity k_n at the origin, then define $i_0 = sgn[p_{1_f}^{(k_n)}(0)]$
(iii) For all other $t = 0, 1, 2, \ldots, l$ let $i_t \in \{-1, 1\}$.

Now we define the set A as

$$A := \begin{cases} i_0, i_1, \ldots, i_l, & \text{if } \; m+n \text{ even} \\ i_0, i_1, \ldots, i_{l-1}, & \text{if } \; m+n \text{ odd.} \end{cases}$$

And lastly, we restate the definition of the imaginary signature associated with an element $I \in A$.

Definition 2. *Let the integers m, n and the functions $q(\omega)$, $q_f(\omega)$ be as already defined. Let $0 = \omega_0 < \omega_1 < \ldots < \omega_{l-1}$ be the real, non-negative, distinct finite zeros of $q_f(\omega)$ with odd multiplicities. Also define $\omega_l = \infty$. For each string $I = \{i_o, i_1, \ldots\} \in A$, let $\lambda(I)$ denote the imaginary signature associated with the string I defined by*

$$\lambda(I) = \begin{cases} \{i_0 - 2i_1 + 2i_2 + \ldots + (-1)^{l-1} 2i_{l-1} & \\ +(-1)^l i_l\}(-1)^{l-1} sgn[q(\infty)], & \text{if } \; m+n \text{ even} \\ \{i_0 - 2i_1 + 2i_2 + \ldots + (-1)^{l-1} 2i_{l-1}\} & \\ \cdot(-1)^{l-1} sgn[q(\infty)], & \text{if } \; m+n \text{ odd.} \end{cases}$$

We will now begin to dissect the constant gain, PI, and PID algorithms presented in [24] for complete characterization of the stabilizing set. These algorithms will be formulated into computational algorithms for implementation onto embedded systems architectures. In most cases, sub algorithms and software libraries will need to be developed to solve sub steps and aid in efficient computation of the results.

2.1 Constant Gain Stabilization

The algorithm to characterize all stabilizing feedback gains can be described as follows. For notational convenience we will refer to this algorithm as the [P] algorithm. The specific details of each step are expanded upon later in section 2.2.

1. Initialization

   a) Find the roots of $N(s)$.

      - Set $lrNs$ equal to the number of roots of $N(s)$ in the OLHP.
      - Set $rrNs$ equal to the number of roots of $N(s)$ in the ORHP.
      - Set $zrNs$ equal to the number of zero roots of $N(s)$.
      - Set $j\omega\_arNs$ equal to the number of purely imaginary roots of $N(s)$.

   b) Set $n = max[deg(D(s)), deg(N(s))]$.

   c) Set $m = deg(N(s))$.

2. Determine the even parts, $Ne(s)$ and $No(s)$, of $N(s)$ and the odd parts, $De(s)$ and $Do(s)$, of $D(s)$.

3. Determine $p_1(\omega)$, $p_2(\omega)$, and $q(\omega)$.

4. Find the subset of roots of $q(\omega)$ that are real, non-negative, distinct, and of odd multiplicity, and order them as $0 = \omega_0 < \omega_1 < \omega_2 < \ldots < \omega_{l-1}$ where l is defined as the cardinality of the subset of roots of $q(\omega)$ found.

5. Determine the set of admissible strings, A.

6. Determine $\gamma(I)$, the imaginary signature of string I.

7. Determine the set $F^* = \{I \in A | \gamma(I) = n - lrNs + rrNs\}$.

8. For each $I_r \in F^*$ find $K_r = (\max_{i_t \in I_r, i_t > 0}[-\frac{p_1(\omega_t)}{p_2(\omega_t)}], \min_{i_t \in I_r, i_t < 0}[-\frac{p_1(\omega_t)}{p_2(\omega_t)}])$, then $K_p = \bigcup_{r=1}^{s} K_r$ where $s = card(F^*)$.

This algorithm is at the core of all the remaining algorithms to be discussed. We will use this algorithm as a basis for developing a functional C++ library for implementing this algorithm onto a Digital Signal Processor (DSP). Because the nature of the problem is concerned with polynomials, we will develop a `Polynomial` class and design specific mathematical functionality to operate on this class of `Polynomial` objects. As a notational convenience we will refer to the steps of this algorithm as [P].1, [P].2, ... , [P].8, the reasons for this will become more clear later when we introduce the algorithms for PI and PID controllers.

2.2 The Polynomial Class

The full class declaration may be found in the appendix. Our focus is implementation onto an embedded system, thus, we do not want to overgeneralize our code base for operations that will never need to be performed. Therefore, we let the algorithm at hand dictate our class functionality. Because we are focusing on embedded DSP, the programming language support will also shape our implementation. TiDSP C++ fully supports the standard C++ libraries, but not much else [25]. We want as little external dependencies in our code as possible; we must adhere to the core of the C++ standards in our design.

The two most important pieces of information a polynomial possesses is its degree and coefficient vector. Thus, our class will define a polynomial in this very simple way. For our purposes the best C++ container to use will be the `std::vector<T>` container class. If our coefficient vector is `v`, then the degree of that `Polynomial` object is simply `v.size()-1`. Thus the basic shell of our `Polynomial` class is presented in Listing 2.1. We set `int degree` and `std::vector<double> coeff` to be private

Listing 2.1: Basic Polynomial Class Declaration

```
class Polynomial {
    public:
        Polynomial(); /* Default Constructors */

        ~Polynomial(); /* Empty Destructor */
    private:
        int degree;
        std::vector<double> coeff;
};
```

data members as to follow best-coding practices. We encapsulated these data members within our class and provided the user of the object functions to interact with them. This protects the object from being used outside of the creators intent and provides reliability and robustness of code. The basic use of this class can be seen in the first three steps of the algorithm. We will need to be able to add, subtract, and multiply two polynomial objects, find the roots of the polynomial, and have access to the polynomials degree and coefficient vector. Using standard operator overloading techniques our class is expanded to the form of Listing 2.2.

Recall that our goal is to not over generalize the code base. To this end, the algorithm dictates specialized mathematical routines to operate on a given polynomial under the demands of this algorithm.

Listing 2.2: Extended Polynomial Class Declaration

```
class Polynomial {
    public:
        Polynomial(); /* Default Constructors */

        std::vector< std::vector<double> > getRoots();

        /*Private Access Functions*/
        int getDegree();
        std::vector<double> getCoeffVec();

        /*Operator Overloads*/
        Polynomial& operator=(const Polynomial &rhs);
        Polynomial& operator+=(const Polynomial &rhs);
        Polynomial& operator-=(const Polynomial &rhs);
        Polynomial& operator*=(const Polynomial &rhs);
        const Polynomial operator+(const Polynomial &other) const;
        const Polynomial operator-(const Polynomial &other) const;
        const Polynomial operator*(const Polynomial &other) const;
        friend Polynomial operator*(double lhs, Polynomial &rhs);

        ~Polynomial(); /* Empty Destructor */

    private:
        int degree;
        std::vector<double> coeff;
};
```

2.2.1 Polynomial Root Finder

A major part of the `Polynomial` class is the ability to compute the roots of a `Polynomial` object. The implementation of the root finder was taken from [26] which is a C++ port of the original FORTRAN algorithm developed by M. A. Jenkins [27]. This FORTRAN and C++ code are both based on the Jenkins-Traub algorithm of [28]. The Jenkins-Traub algorithm is a near standard in the field of numerical computation of polynomial roots. It is a three-stage, extremely effective, globally convergent algorithm designed specifically for computing the roots of polynomials.

For this reason it is the definitive choice for implementation onto an embedded system.

2.2.2 Even and Odd Parts of a Polynomial

In step [P].2 we require an even and odd decomposition of the polynomials $N(s)$ and $D(s)$. For any polynomial $P(s)$ we can define a decomposition into even and odd parts, $P_e(s^2)$ and $P_o(s^2)$ respectively, where $P(s) = P_e(s^2) + sP_o(s^2)$.

Ex. 1 — Let $P(s) = s^5 + 11s^4 + 22s^3 + 60s^2 + 47s + 25$. Identify $P_e(s)$ and $P_o(s)$.

Answer (Ex. 1) —

$$P_e(s^2) = 11s^4 + 60s^2 + 25$$
$$P_o(s^2) = s(s^4 + 22s^2 + 47)$$

Ex. 2 — Let $P(s) = s^4 + 6s^3 + 12s^2 + 54s + 16$. Identify $P_e(s)$ and $P_o(s)$.

Answer (Ex. 2) —

$$P_e(s^2) = s^4 + 12s^2 + 16$$
$$P_o(s^2) = s(s^2 + 54)$$

To computationally solve this simple analytical problem we prescribe the procedures of Algorithms 1 and 2.

Algorithm 1 Polynomial::getEvenPart()

Precondition: P is the polynomial being operated on

```
function getEvenPart()
    if deg($P$) is odd then
        Remove the leading coefficient of $P$
    Set the odd exponent coefficients of $P$ to zero
```

Algorithm 2 Polynomial::getOddPart()

Precondition: P is the polynomial being operated on

```
function getOddPart()
    q ← deg($P$)
    Remove the last coefficient of $P$
    if deg($P$) = -1 then
        Return 0
    if q = 0 then
        Remove the leading coefficient of $P$
    Set the odd exponent coefficients of $P$ to zero
```

2.2.3 Conversion of the s Parameter to $j\omega$

In step [P].3 it is necessary to convert the polynomials $p_1(s) \to p_1(w)$, $p_2(s) \to p_2(w)$, and $q(s) \to q(w)$. This is done by replacing the parameter s with $j\omega$. This is a conversion from the s-domain to the frequency domain, and while analytically this is easy to accomplish, computationally the process may not be as straightforward. We first consider an example.

Ex. 3 — Let $P(s) = 5s^8 + 6s^6 - 549s^4 - 1278s^2 + 400$. Evaluate $P(\omega)$.

Answer (Ex. 3) —

$$\begin{aligned} P(\omega) &= 5(j\omega)^8 + 6(j\omega)^6 - 549(j\omega)^4 - 1278(j\omega)^2 + 400 \\ &= 5\omega^8 - 6\omega^6 - 549\omega^4 + 1278\omega^2 + 400 \end{aligned}$$

Ex. 4 — Let $P(s) = s^6 + 5s^5 + 3s^4 - 10s^3 + 2s^2 + 1$. Evaluate $P(\omega)$.

Answer (Ex. 4) —

$$\begin{aligned} P(\omega) &= (j\omega)^6 + 5(j\omega)^5 + 3(j\omega)^4 - 10(j\omega)^3 + 2(j\omega)^2 + 1 \\ &= -\omega^6 + 5j\omega^5 + 3\omega^4 + 10j\omega^3 - 2\omega^2 + 1 \end{aligned}$$

To make the algorithm less complex we exploit the nature of the problem. An s-domain to frequency domain conversion only appears in step [P].3 where we concentrate on $p_1(s)$, $p_2(s)$ and $q(s)$. The terms $p_1(s)$ and $p_2(s)$ are polynomials with only even powered terms. This is easy to see by the definitions in [24],

$$\begin{aligned} p_1(s) &= D_e(s^2)N_e(s^2) - s^2 D_o(s^2)N_o(s^2) \\ p_2(s) &= N_e^2(s^2) - s^2 N_o^2(s^2). \end{aligned}$$

Thus, replacing s with $j\omega$ produces $(j\omega)^e$ terms, where e is an even integer. Since $(j\omega)^e = j^e\omega^e$ and $j^e = \pm 1$ we can determine the value of j^e based on e. That is

$$j^e = \begin{cases} 1, & \text{if } \frac{e}{2} \text{ is even} \\ -1, & \text{if } \frac{e}{2} \text{ is odd.} \end{cases}$$

Therefore, the sign of the coefficient as a result of the domain conversion can be easily calculated by Algorithm 3.

Algorithm 3 Polynomial::stojw()

Precondition: P is the polynomial being operated on, with coefficient vector cv, 0-indexed with the 0 index being the highest powered term of P

```
function STOJW()
    for $i \leftarrow 0$ to $deg(P) - 1$ step 2 do
        $cv[i] \leftarrow cv[i](((deg(P) - i)/2 \bmod 2) \times -2 + 1)$
```

However, for $q(s) = s(N_e(s^2)D_o(s^2) - D_e(s^2)N_o(s^2)$ we have terms of the form $(j\omega)^d$ where d is an odd integer. This leads to two issues when we consider $(j\omega)^d = j^d\omega^d$. First, the term j^d will be imaginary because d is odd. However, the core of the algorithms computation is only concerned with $q(s)|_{j\omega}$ where $q(s)|_{j\omega} = jq(\omega)$. Thus, we only consider the terms j^{d-1} where $d - 1$ is even and we have resolved this previously. The second issue is that all the terms of the polynomial are odd integer powers where in the case of $p_1(s)$ and $p_2(s)$ they were even. In order to reuse the algorithm developed for the $s \to j\omega$ conversion of $p_1(s)$ and $p_2(s)$ and apply it to $q(s)$ without making the algorithm case dependent or worse, writing an entirely separate function, we make a small modification. Unlike $p_1(s)$ and $p_2(s)$, the term $q(s)$ will be composed of odd powered terms due to the extra factor of s. That is, we had even powered terms s^e for $p_1(s)$ and $p_2(s)$, but now we have odd powered terms s^{e+1} for $q(s)$. We originally considered $\frac{e}{2}$ to determine the sign of the coefficient for conversion. Now, we will look at $\lfloor \frac{e}{2} \rfloor$. Since $\lfloor \frac{e}{2} \rfloor = \frac{e}{2} = \lfloor \frac{e+1}{2} \rfloor$ we avoid the discrepancy of the $q(s)$ term and have a single algorithm which returns the proper coefficient values for the real polynomials $p_1(\omega)$, $p_2(\omega)$, and $q(\omega)$ when given $p_1(s)$, $p_2(s)$ and $q(s)$. The unified algorithm is shown in Algorithm 4.

Algorithm 4 Polynomial::stojw()

Precondition: P is the polynomial being operated on, with coefficient vector cv, 0-indexed with the 0 index being the highest powered term of P

```
function STOJW()
    for $i \leftarrow 0$ to $deg(P) - 1$ step 2 do
        $cv[i] \leftarrow cv[i]((\lfloor(deg(P) - i)/2\rfloor \bmod 2) \times -2 + 1)$
```

2.2.4 The n^{th} Derivative of a Polynomial Evaluated at Zero

In step [P].5 of the algorithm, [24] defines the term A as being the set of all possible strings $s = \{i_0, i_1, i_2, \ldots\}$, $i_r \in \{0, \pm 1\}$ of length l or $l-1$ depending on the value of $m+n$. This definition requires special attention in regards to $N^*(j\omega_t)$ and $N^*(s)$. For each root, ω_t of $q(w)$ found in step [P].4 we check $N^*(j\omega_t)$ for some $t = 1, 2, \ldots, l-1$. If $N^*(j\omega_t) = 0$ then $i_t = 0$ and if $N^*(s)$ has zeros of multiplicity k_n at the origin, then $i_0 = sgn[p_{1_f}^{(k_n)}(0)]$.

By definition, $N^*(s) = N(-s)$, which means that a root of $N^*(s)$ is also a root of $N(s)$. The roots of $N(s)$ were found in step [P].1. Thus, if $zrNs > 0$ then $N^*(s)$ has a zero of multiplicity $zrNs$ at the origin and we must compute $sgn[p_{1_f}^{(k_n)}(0)]$, where $p_{1_f}(\omega) = \frac{p_1(\omega)}{(1+\omega^2)^{\frac{m+n}{2}}}$. If we examine the derivatives of $p_{1_f}(\omega)$ evaluated at 0, we will find the following pattern emerge:

Let $p(\omega) = a_\nu \omega^\nu + a_{\nu-1}\omega^{\nu-1} + \ldots + a_1\omega + a_0$, then

$$p_f^{(k_n)}(0) = \frac{k_n!}{0!}a_{k_n} - \frac{k_n!}{1!}a_{k_n-2}(\frac{m+n}{2}) + \frac{k_n!}{2!}a_{k_n-4}(\frac{m+n}{2})(\frac{m+n}{2}+1) - \frac{k_n!}{3!}a_{k_n-6}(\frac{m+n}{2})(\frac{m+n}{2}+1)(\frac{m+n}{2}+2) + \ldots$$

From this pattern a simple algorithm can be constructed to easily compute $p_{1_f}^{(k_n)}(0)$ for any k_n without complicated derivative calculating methods.

The second condition, $N^*(j\omega_t)$, will only hold if $N^*(s)$ has a root at the origin or equivalently, if $N(s)$ has a root at the origin. Thus, we only check the $N^*(j\omega_t) = 0$ condition if $N(s)$ has a root at the origin. The check is done by evaluating $N^*(s)$ at the purely imaginary value $j\omega_t$ and checking if it is 0. If it is 0 then we set the t^{th} column of the A matrix to 0.

2.2.5 Limits of Polynomials

In step [P].6 we calculate the imaginary signature [24] for each string in A. The imaginary signature is defined as

$$\lambda(I) = \begin{cases} \{i_0 - 2i_1 + 2i_2 + \ldots + (-1)^{l-1}2i_{l-1} & \\ +(-1)^l i_l\}(-1)^{l-1} sgn[q(\infty)], & \text{if } \ m+n \text{ even} \\ \{i_0 - 2i_1 + 2i_2 + \ldots + (-1)^{l-1}2i_{l-1}\} & \\ \cdot(-1)^{l-1} sgn[q(\infty)], & \text{if } \ m+n \text{ odd.} \end{cases}$$

which is computationally straightforward except for the $sgn[q(\infty)]$ term. This is much easier to solve than it appears. The $sgn[q(\infty)]$ means $sgn[\lim_{\omega\to\infty} q(\omega)]$ and since $q(w)$ is a polynomial, its limit is defined by the leading coefficient. Moreover, we are only concerned with the sign of that limit. For any polynomial

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0,$$

it can be shown that

$$\lim_{x\to\infty} p(x) = \begin{cases} +\infty, & \text{if } sgn[a_n] = 1 \\ -\infty, & \text{if } sgn[a_n] = -1. \end{cases}$$

For our purposes we define the $sgn[+\infty] = 1$ and $sgn[-\infty] = -1$, thus, the $sgn[\lim_{\omega\to\infty} q(\omega)] = sgn[a_n]$. Therefore, we define a special function that quickly reads the value of the leading coefficient of $q(\omega)$ and computes its sign.

2.2.6 The Limit of the Quotient of Two Polynomials

In step [P].8 of the algorithm it is required that $-\frac{1}{G(j\omega_t)} = -\frac{p_1(\omega_t)}{p_2(\omega_t)}$ be evaluate at each ω_t value. This is simple enough because our `Polynomial` class can easily evaluate polynomials given a real valued input. However, the trouble occurs when $m+n$ is even. As covered in [24] when $m+n$ is even $\omega_l = \infty$ and thus we are required to evaluate $-\frac{p_1(\infty)}{p_2(\infty)}$ or more exactly $\lim_{\omega\to\infty} -\frac{p_1(\omega_t)}{p_2(\omega_t)}$. Again, we exploit the nature of our problem to make the computation much simpler. In the space of polynomials we have that

$$\lim_{x\to\infty} \frac{\sum_{i=0}^{m} a_i x^i}{\sum_{k=0}^{n} b_k x^k} = \begin{cases} 0, & \text{if } m < n, (a_m, b_n) \neq 0 \\ \text{doesn't exist}, & \text{if } m > n, (a_m, b_n) \neq 0 \\ \frac{a_m}{b_n}, & \text{if } m = n, (a_m, b_n) \neq 0. \end{cases}$$

Therefore, a simple check of the degrees of $p_1(\omega)$ and $p_2(\omega)$ will give us a simple means of computing this limit.

2.3 Characterization of All Stabilizing PI Controllers

Using the bare essentials of the mechanics of computation, we again break down the algorithm that defines the entire set of stabilizing PI parameters for a PI controller found in [24]. The specific details of each step will be expanded upon later if necessary.

1. Initialization

    a) Find the roots of $N(s)$.

    - Set $lrNs$ equal to the number of roots of $N(s)$ in the OLHP.
    - Set $rrNs$ equal to the number of roots of $N(s)$ in the ORHP.
    - Set $zrNs$ equal to the number of zero roots of $N(s)$.
    - Set $j\omega\_arNs$ equal to the number of purely imaginary roots of $N(s)$.

   b) Set $n = max[deg(D(s)), deg(N(s))] + 1$.

   c) Set $m = deg(N(s))$.

2. Determine the even parts, $Ne(s)$ and $No(s)$, of $N(s)$ and the odd parts, $De(s)$ and $Do(s)$, of $D(s)$.

3. Determine $p_1(\omega)$, $p_2(\omega)$, $q_1(\omega)$, and $q_2(\omega)$ where

$$p(\omega, k_i) = p_1(\omega) + k_i p_2(\omega)$$
$$q(\omega, k_p) = q_1(\omega) + k_p q_2(\omega).$$

4. Narrow down the sweeping interval for K_p via root locus ideas.

   a) Define $U(\omega) = \frac{1}{\omega} q_1(\omega)$ and $V(\omega) = \frac{1}{\omega} q_2(\omega)$.

   b) Solve $\frac{d}{dw}\left(\frac{V(\omega)}{U(\omega)}\right) = 0$.

   c) Solve for k_i using each root from step 4b where $k_i = -\frac{U(\omega_i)}{V(\omega_i)}$.

   d) Order the k_i values from smallest to largest.

   e) Generate intervals from the set of ordered k_i.

    - If no root of step 4b is 0 then define $k_p^* = \frac{U(0)}{V(0)}$.
    - If $k_p^* \in (k_i, k_{i+1})$ then make (k_i, k_p^*) and (k_p^*, k_{i+1}) intervals.

f) Choose[1] a value in each interval, k_p', and find the real roots of $U(\omega) + k_p' V(\omega) = 0$ and let l' be the count of real and positive roots.

g) A valid K_p interval is found if

$$\begin{cases} l' + 1 \geq \frac{1}{2}|n - (lrNs - rrNs)|, & \text{if } \ m + n \text{ even} \\ l' + 1 \geq \frac{1}{2} + \frac{1}{2}|n - (lrNs - rrNs)|, & \text{if } \ m + n \text{ odd.} \end{cases}$$

5. For each k_p in the valid ranges of step [PI].4 we set $q_f(\omega) = \frac{q_1(\omega)+k_p q_2(\omega)}{(1+\omega^2)^{\frac{m+n}{2}}}$ and solve for the K_i interval through the [P] algorithm beginning at step [P].4.

To implement the [PI] algorithm above we will build on the [P] algorithm already established. We see this in step [PI].5 where for each k_p value we run the [P] algorithm to find the valid range of k_i values. Further, steps [PI].1, [PI].2, and [PI].3 are much the same as [P].1, [P].2, and [P].3. The major addition to the [PI] algorithm is that in step [PI].4, the sweeping interval for K_p is narrowed down. The analytical theory behind this step is well covered in the appendix of [24]. To create the computational implementation of this procedure we make use of the bare essentials when considering each of these analytical steps.

2.3.1 Dividing $q_1(\omega)$ and $q_2(\omega)$ by ω

Because of the nature of our problem $q_1(\omega)$ and $q_2(\omega)$ will never contain a constant term. Thus, when defining $U(\omega)$ and $V(\omega)$, the division of the polynomials $q_1(\omega)$ and $q_2(\omega)$ by ω is equivalent to removing the last element from the coefficient vector. Thus a special function was made to delete this element and decrease the degree of the polynomial by 1.

[1]The choice of this value is important because the interval condition is not necessary. Thus an expansion search method is used to find the correct value on the infinite intervals

2.3.2 Derivative of the Quotient of Two Polynomials

One important part of narrowing the K_p range involves the computation of the roots of $\frac{d}{dw}\left(\frac{V(\omega)}{U(\omega)}\right) = 0$. Using the quotient rule for derivatives we have $\frac{UV'-VU'}{U^2} = 0$. Because we are only concerned with the roots of this derivative, we only need to be concerned with the numerator, $UV' - VU'$. We can simplify the computation by considering only the equation $UV' - VU' = 0$. Since $U(\omega)$ and $V(\omega)$ are both polynomials, their first derivatives are easy to find; a simple shift of the coefficient vector and multiplication by its previous index value. A special function is made to perform this process. Given the $U(\omega)$ and $V(\omega)$ polynomials, $UV' - VU'$ is computed and returned, ready to be passed into the root finder.

2.3.3 Determining the Valid K_p Intervals

We must pay careful attention to the infinite intervals as it pertains to the choice of k_p' in step [PI].4f. The root locus theory of [24] provides the following condition for the finite intervals: if $U(0) + k_pV(0) \neq 0$ for all $k_p \in (k_i, k_{i+1})$, then the distribution of the real roots of $U(\omega) + k_pV(\omega) = 0$ with respect to the origin is invariant over this range of k_p values. However, this does not apply for the infinite intervals of $(-\infty, k_1)$ and $(k_l, +\infty)$. For this reason we implement an exponential expansive search algorithm. We know how many real and positive roots are needed in the interval for it to be a valid range narrowing interval. By checking k_1^- and k_l^+ we can easily examine the roots of $U(\omega) + k_1^-V(\omega) = 0$ and $U(\omega) + k_l^+V(\omega) = 0$, to see if this entire interval will provide more information on the K_p range. If it will, then we choose another point that is twice the distance away from k_1 and k_l as k_1^- and k_l^+. We continue this exponential expansion until the number of roots fails to satisfy the necessary number of roots to be a valid interval. Then by performing a binary search over the interval formed by the previous two points checked, we can find the point to form a finite interval around and define a valid K_p range from the infinite interval.

2.3.4 A "non-fragile" Controller Via Center of Mass

In [24] the concept of a "non-fragile" controller is discussed in relation to PID controllers. However, a similar treatment is not found for PI controllers. We present here a methodology similar to finding the optimal "non-fragile" controller for PID but applied to PI. In PID control the stabilizing sets of (k_i, k_d) for a given k_p value were defined by linear convex polygons. To this end an LP problem was formulated to solve the largest inscribed circle. However, with PI control the stabilizing regions do not necessarily have linear boundaries and curve fitting would be a computational burden. A different approach is needed.

In order to determine a close approximation to the center of an irregular convex region in $\mathbb{R}^2$ we employ a center of mass calculation on the space. Suppose that a plate is bounded by two curves $y_0 = f(x)$ and $y_1 = g(x)$ in $\mathbb{R}^2$ on the interval $[a, b]$ with $y_0 > y_1 \forall x$. In [29] the mass of the plate with area A is defined as

$$\begin{aligned} M &= \rho A \\ &= \rho \int_a^b (f(x) - g(x))\, dx \end{aligned}$$

and the moments of the plate along the x and y region as

$$\begin{aligned} M_x &= \rho \int_a^b \frac{1}{2} \left([f(x)]^2 - [g(x)]^2\right) dx \\ M_y &= \rho \int_a^b x\,(f(x) - g(x))\, dx. \end{aligned}$$

The coordinates of the center of mass, $(\bar{x}, \bar{y})$, are given as

$$\bar{x} = \frac{M_y}{M} = \frac{\int_a^b x(f(x) - g(x))dx}{\int_a^b f(x) - g(x)dx} = \frac{1}{A}\int_a^b x(f(x) - g(x))dx$$
$$\bar{y} = \frac{M_x}{M} = \frac{\int_a^b \frac{1}{2}\left([f(x)]^2 - [g(x)]^2\right)dx}{\int_a^b f(x) - g(x)dx} = \frac{1}{A}\int_a^b \frac{1}{2}\left([f(x)]^2 - [g(x)]^2\right)dx.$$

We assume that the mass of the shape is uniform over its surface, and that the region is defined by two curves. In our implementation we have a sampling of these two curves. Thus, we must cast the formulation into a discrete space. For simplicity we will first assume that $\rho = 1$. This is allowed because ρ represents the density of the physical disc, but since there is no real physical shape we are free to make this any value. For each $k_{p_i} \in K_p$ let the associated K_i interval have n points and be defined as $K_i = \{k_{i_0}, k_{i_1}, \ldots, k_{i_{n-1}}\}$. Also let the K_p interval have m points and be defined as $K_p = \{k_{p_0}, k_{p_1}, \ldots, k_{p_{m-1}}\}$. Using Riemann integrals we define

$$\Delta x = k_{p_j} - k_{p_{j-1}}$$
$$l_{K_i} = k_{i_{n-1}} - k_{i_0}$$
$$M = A = \sum_{j=1}^{m-1} |l_{K_i}|\Delta x.$$

The coordinates of the center of mass, $(\bar{x}, \bar{y})$, are given as

$$\bar{x} = \frac{1}{M}\sum_{j=1}^{m-1} k_{p_j}|l_{K_i}|\Delta x$$
$$\bar{y} = \frac{1}{2M}\sum_{j=1}^{m-1} (k_{i_{n-1}}^2 - k_{i_0}^2)sgn(l_{K_i})\Delta x$$

where we use the $sgn(l_{K_i})$ to adjust for which curve is above or below the other.

This formulation is computationally based on the data already generated and purely mathematical, making it easy to implement into our embedded systems algorithm.

For convex sets, it is guaranteed that the center of mass will exist within the set. However, for disjoint sets, if we treat them as one set, the center of mass is not guaranteed to exist within either one of the sets. Disjoint sets will occur if there is a disjoint interval in the K_p set. To overcome this problem we must treat each disjoint set as a separate object when calculating the center of mass. Then, a method for determining the best overall center of mass should be implemented.

2.4 Characterization of All Stabilizing PID Controllers

The characterization of all k_p, k_i, and k_d values of a PID controller for a given plant model is extended from the [PI] algorithm covered previously. The major addition to this algorithm will be the need for a Linear Program Solver [30] for a series of constrained optimization problems and a 2D space quantization algorithm. The algorithm is detailed below.

1. Initialization

   a) Find the roots of $N(s)$ and define $lrNs$, $rrNs$, $zrNs$, and jw_arNs as before.

   b) Set $n = max[deg(D(s)) + 1, deg(N(s)) + 2]$.

   c) Set $m = deg(N(s))$.

2. Determine the even parts, $Ne(s)$ and $No(s)$, of $N(s)$ and the odd parts, $De(s)$ and $Do(s)$, of $D(s)$.

3. Determine $p_1(\omega)$, $p_2(\omega)$, $q_1(\omega)$, and $q_2(\omega)$ where

$$\begin{aligned} p(\omega, k_i, k_d) &= p_1(\omega) + (k_i - k_d\omega^2)p_2(\omega) \\ q(\omega, k_p) &= q_1(\omega) + k_p q_2(\omega). \end{aligned}$$

4. Narrow down the sweeping interval for K_p. (See step [PI].4)

5. For each k_p in the valid ranges of step [PID].4 we set $q_f(\omega) = \frac{q_1(\omega)+k_p q_2(\omega)}{(1+\omega^2)^{\frac{m+n}{2}}}$ and solve steps[P].4 - [P].7. This produces an $F^*_{k_p}$ for each k_p.

6. Solve for k_i and k_d values for each k_p via Theorem 4.4.1 of [24].

The major change from the [PI] to the [PID] algorithm is in step [PID].6. This step is based on the results of Theorem 4.4.1 of [24] in which the following set of linear inequalities is defined, $p(\omega_t, k_i, k_d)i_t > 0$. This means that for each root, ω_t, of $q_f(\omega)$ we have[2]

$$\begin{aligned} p(\omega_t, k_i, k_d)i_t > 0 &\Rightarrow [p_{1_f}(\omega_t) + (k_i - k_d\omega_t^2)p_{2_f}(\omega_t)]i_t > 0 \\ &\Rightarrow p_{1_f}(\omega_t)i_t + k_i p_{2_f}(\omega_t)i_t - k_d\omega_t^2 p_{2_f}(\omega_t)i_t > 0 \\ &\Rightarrow k_i p_{2_f}(\omega_t)i_t - k_d\omega_t^2 p_{2_f}(\omega_t)i_t > -p_{1_f}(\omega_t)i_t \\ &\Rightarrow \underbrace{\begin{bmatrix} p_{2_f}(\omega_t)i_t & -\omega_t^2 p_{2_f}(\omega_t)i_t \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} k_i \\ k_d \end{bmatrix}}_{x} \le \underbrace{\begin{bmatrix} p_{1_f}(\omega_t)i_t \end{bmatrix}}_{b} \end{aligned}$$

This set of linear inequalities, $Ax \le b$, forms a convex set. As described in [24], one can choose an optimal (k_p, k_i, k_d) triplet by defining the largest inscribed circle

[2]We chose to derive the negation of the formula presented in Theorem 4.4.1 so that it would fit better into the constrained optimization problem of finding the largest inscribed circle of a convex polygon.

of the convex set and taking the center of this circle to be the optimal result. The aim of such a method is to choose the stabilizing set of points that exists furthest away from the stability boundaries. This guarantees the best stability performance against perturbations in the controller parameters. To do this we define the convex set $P = \{x | a_i^T x \leq b_i\}$ and the circle $C = \{x_c + u | \|u\| \leq r\}$ where x_c is the circle's center and r is the circle's radius. Then the inscribed circle constraint is as follows

$$sup\{a_i^T x_c + a_i^T u \mid \|u\| \leq r\} \leq b_i \Rightarrow a_i^T x_c + r\|a_i\| \leq b_i \ .$$

Thus, we can establish an LP problem of the following form

$$\begin{aligned} &\text{Maximize} \quad r \\ &\text{Subject to} \ \ a_i^T x_c + r\|a_i\| \leq b_i \ . \end{aligned}$$

2.4.1 The Linear Program Solver

To solve the LP problem of the largest inscribed circle we divide the problem into two classes. Class 1 involves the set of constraints that form a triangular shaped feasibility region (i.e. a three vertex feasibility region) and class 2 involves the the set of constraints that form a convex polygon of four or more vertices.

To solve Class 1 problems we exploit the nature of the geometry of the feasibility region of three vertices. The incenter [31] of a triangle is the center point of the largest inscribed circle of the triangle. The incenter can be easily calculate when given the coordinates of the triangles vertices. Let the three vertices be located at (x_a, y_a), (x_b, y_b), and (x_c, y_c), with the opposite sides to these vertices having lengths

a, b, and c, respectively, then the incenter is define as

$$\left(\frac{ax_a + bx_b + cx_c}{P}, \frac{ay_a + by_b + cy_c}{P}\right)$$

where $P = a + b + c$. Thus, to save computation time we apply this calculation if the feasible region is a triangle, instead of formulating the LP problem and using a computationally intensive LP Solver.

To solve the Class 2 problems we require a computational LP solver. A popular open source variety LP solver can be found at [32]. This solver is based on the revised simplex method[33] and the branch-and-bound method[34]. Full C++ source code is available for integration into any programming projects under the GNU Lesser General Public License. This package will accurately solve the constraint optimization problem at hand. However, to make this package work under TiDSP C++ some modifications had to be made to the source code. Because the package is not implemented with embedded systems in mind, there is a large external dependency in the code to a `WIN32` environment. Fortunately, this dependence is only tied to the more aesthetical reporting and logging features found within the package. Stripping all the "unnecessary" formatting, logging, and reporting features of this package and leaving only the basic necessary mathematical computation code yields a smaller code set that is fully functional within a TiDSP C++ environment. We employ the use of this open source package when solving the LP problem of the largest inscribed circle for a convex set of four or more vertices.

2.4.2 Finding the Feasibility Region

The convex set defined by $Ax \leq b$ is called the feasibility region. Given a specific k_p value, a convex 2D shape is defined in the (k_i, k_d) plane. This convex shape (typically a triangle or quadrilateral) is the entire set of (k_i, k_d) pairs that will stabilize the system for a given k_p.

In the last section we divided the LP problem into two classes, which required knowledge of the number of vertices in the feasibility region. This information is not immediately apparent from the set of linear inequalities defining the region itself. Our main goal is to determine the set of cartesian coordinates that define the corners of the convex set, also known as the vertex enumeration problem [35]. While efficient algorithms to solve this problem exist [36], a naive implementation was done following Algorithm 5.

Algorithm 5 Naive Vertex Enumeration: A brute force approach to finding the set of vertices which belong to the corners of a region of R^2 defined by a set of linear inequalities.

```
function FEASIBLE_REGION()
Find all intersection points of $Ax = b$
For each vertex, test the point $v_i$ against $Av_i \leq b$
If the vertex passes all inequalities tests, then it is a valid vertex
Those vertices which satisfy all inequalities of $Ax \leq b$ constitute the set we want
```

To determine the intersection of two lines we follow a basic mathematical formulation using Cramer's rule. Given a system of equations that defines two lines, $\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} x = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$ then the intersection point is defined as

$$x_i = \frac{b_1 a_4 - b_2 a_2}{a_1 a_4 - a_2 a_3}$$
$$y_i = \frac{b_2 a_1 - b_1 a_3}{a_1 a_4 - a_2 a_3} \ .$$

2.4.3 Quantizing the Convex Set

For the purpose of exhaustive search over the entire space of (k_p, k_i, k_d) triplets for a PID controller design with an optimal performance criterion, it is necessary to quantize the convex space into individual (k_p, k_i, k_d) data points. The resolution of the quantization is a parameter of a special function developed to perform this specific task. A higher resolution of the quantization provides a better estimate of the optimal solution but increases search time, while a lower resolution produces poorer optimality but will decrease the search time. The computational algorithm to accomplish this is long and complex. However, the basic idea is to first develop a convex hull ordering of the vertices. Then we quantize the x-axis on the interval from the least x-coordinate to the greatest x-coordinate. For each quantization level along the x-axis, we quantize the y-axis from the lower line to the upper line. This is very similar to a discrete form of two-variable calculus when determining the area of a non uniform shape in $\mathbb{R}^2$.

Remark: For the purposes of characterizing the entire set of stabilizing k_p, k_i, and k_d values of a PID controller, the LP solver, defined feasibility region, and quantization of the region is not necessary. The set of linear inequalities created in step [PID].6 completely defines the entire set. The LP solver exists only to find the center of the largest inscribed circle over the entire space. The vertex enumeration solver only exists for the purpose of quantizing the space; with the added benefit of easing computation if the convex set forms a triangle. And, quantization of the space is only necessary for searching over the controller design space to satisfy a specific optimality criterion. It is not our intent to convey that the added functionality is necessary to the core of the algorithm, however, the LP solver, defined feasibility region, and quantization of the feasibility region are of practical importance.

2.4.4 Correcting for the Pure Derivative Term

Under PID control, the pure derivative term is undesirable for two reasons: amplification of high frequency noise, and the closed loop system may be internally unstable. To fix this undesirable symptom we instead use the controller structure

$$C(s) = k_p + \frac{k_i}{s} + \frac{k_d s}{1 + Ts}$$

where T is a small positive number usually between 0.01 and 0.1. This new controller will modify the closed loop characteristic polynomial that the [PID] algorithm is based on. However, in [24] it is shown that by a single linear transformation of the terms of the closed loop characteristic polynomial, the polynomial can be put into the same form as required by the original [PID] algorithm. Then, a solution can be formed via the methods already discussed and transformed back. This transformation is defined as

$$\begin{bmatrix} k_p \\ k_i \\ k_d \end{bmatrix} = \begin{bmatrix} 1 & -T & 0 \\ 0 & 1 & 0 \\ -T & T^2 & 1 \end{bmatrix} \begin{bmatrix} k_p' \\ k_i' \\ k_d' \end{bmatrix}$$

Instead of performing matrix-vector multiplication we consider the major computational pieces of this transformation. That is

$$\begin{aligned} k_p &= k_p' - Tk_i' \\ k_i &= k_i' \\ k_d &= -Tk_p' + T^2 k_i' + k_d'. \end{aligned}$$

We immediately realize that k_i doesn't require a transformation. Thus these two basic computations are easier to perform and faster than a full matrix-vector multiplication implementation.

2.5 Examples and Results

In this section we verify all the examples in [24] for the [P], [PI], and [PID] algorithms presented in this chapter. In all cases we consider the following plant model

$$G(s) = \frac{N(s)}{D(s)}$$

$$\begin{aligned} N(s) &= a_m s^m + a_{m-1} s^{m-1} + \cdots + a_1 s + a_0 \\ D(s) &= b_n s^n + b_{n-1} s^{n-1} + \cdots + b_1 s + b_0 \end{aligned}$$

where $n \geq m$.

2.5.1 Constant Gain Stabilization

Consider the plant defined as

$$\begin{aligned} N(s) &= s^4 + 6s^3 + 12s^2 + 54s + 16 \\ D(s) &= s^5 + 11s^4 + 22s^3 + 60s^2 + 47s + 25\ . \end{aligned}$$

When executing the [P] algorithm for this specific plant on our embedded system we get the following results.

$$p_1(\omega) = 5.0\omega^8 - 6.0\omega^6 - 549.0\omega^4 + 1278.0\omega^2 + 400$$
$$p_2(\omega) = 1.0\omega^8 + 12.0\omega^6 - 472.0\omega^4 + 2532.0\omega^2 + 256.0$$
$$q(\omega) = 1.0\omega^9 + 32.0\omega^7 - 627.0\omega^5 + 2474.0\omega^3 - 598.0\omega$$

The real, non-negative, distinct, and finite zeros of $q(\omega)$ with odd multiplicity are

$$\omega_0 = 0.000000$$
$$\omega_1 = 0.508343$$
$$\omega_2 = 2.417352$$
$$\omega_3 = 2.915147$$

The set F^* is

$$\begin{Bmatrix} (1,-1,-1,1) \\ (1,1,-1,-1) \\ (1,1,1,1) \end{Bmatrix}$$

which produces the final stabilizing set of constant gain controllers as

$$k_p \in (-0.788981, 2.503451) \cup (22.493895, \infty) \ .$$

It is impossible to verify the stability of the system over all possible k_p values, but as a representative example to the systems stability and correctness of the algorithm we consider three different k_p values in the K_p interval, and simulate the system using the MATLAB SIMULINK model of Figure 2.1. The step response plot can be seen in Figure 2.2 where it is verified to be stable.

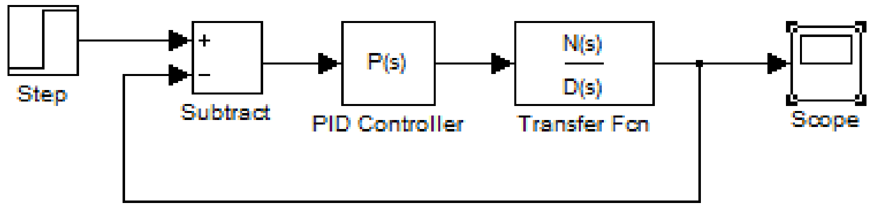

Figure 2.1: Simulation model for P-control of an LTI system.

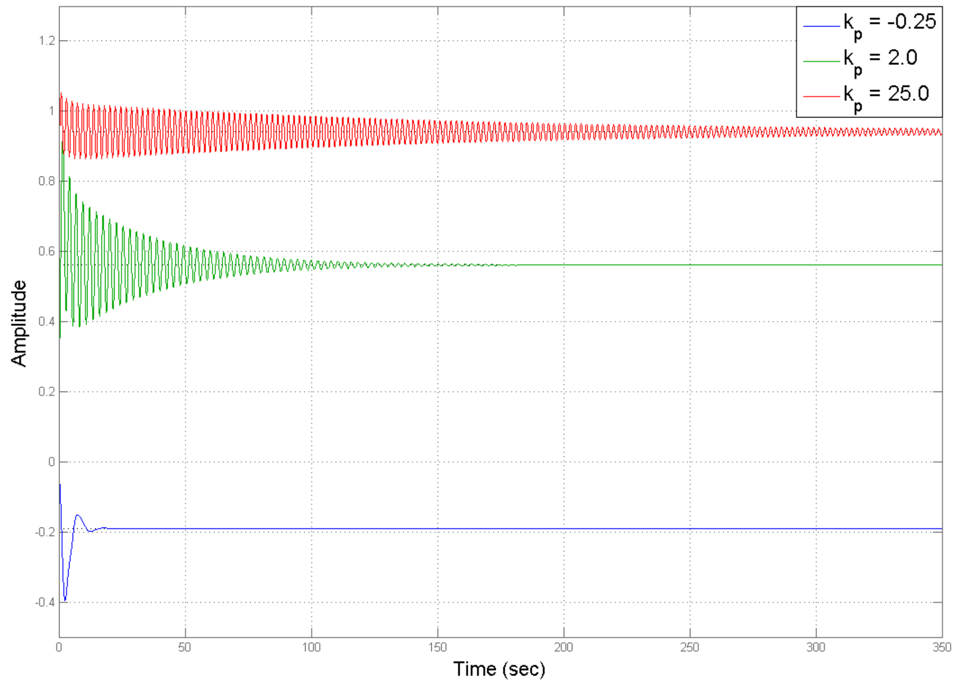

Figure 2.2: Step response with $k_p = \{-0.25, 2.0, 25.0\}$.

As a second example, consider the plant defined as

$$N(s) = s^5 + 2s^4 + 3s^2 + 4s + 1$$

$$D(s) = s^5 - 2s^4 + 3s^3 + 7s^2 + 10s + 7.0 \ .$$

When executing the [P] algorithm for this specific plant on our embedded system we get the following results.

$$p_1(\omega) = 1.0\omega^{10} - 7.0\omega^8 + 6.0\omega^6 + 21.0\omega^4 + 12.0\omega^2 + 7$$
$$p_2(\omega) = 1.0\omega^{10} + 4.0\omega^8 - 4.0\omega^6 + 13.0\omega^4 + 10.0\omega^2 + 1.0$$
$$q(\omega) = 4.0\omega^9 - 2.0\omega^7 + 31.0\omega^5 - 5.0\omega^3 - 18.0\omega$$

The real, non-negative, distinct, and finite zeros of $q(\omega)$ with odd multiplicity are

$$\omega_0 = 0.000000$$
$$\omega_1 = 0.911462$$

The set F^* is

$$\Big\{(-1, 1, -1)\Big\}$$

which produces the empty set, $k_p \in \varnothing$, meaning that no constant gain controller will stabilize this plant. The results of these examples verify correctness of the [P] algorithm implemented on the embedded system because they match the results found in [24].

2.5.2 Stabilization Using a PI Controller

Consider the plant defined as

$$N(s) = s^3 + 6s^2 - 2s + 1$$
$$D(s) = s^5 + 3s^4 + 29s^3 + 15s^2 - 3s + 60\ .$$

When executing the [PI] algorithm for this specific plant on our embedded system we get the following results:

$$
\begin{aligned}
p_1(\omega) &= 3.0\omega^8 - 166.0\omega^6 - 19.0\omega^4 - 117.0\omega^2 \\
p_2(\omega) &= 1.0\omega^6 + 40.0\omega^4 - 8.0\omega^2 + 1.0 \\
q_1(\omega) &= -1.0\omega^9 + 9.0\omega^7 + 154.0\omega^5 - 369.0\omega^3 + 60.0\omega \\
q_2(\omega) &= 1.0\omega^7 + 40.0\omega^5 - 8.0\omega^3 + 1.0\omega \ .
\end{aligned}
$$

Performing the K_p narrowing routine in this example produces the finite K_p sweeping interval to be

$$k_p \in (-2.541190, 16.443085) \ .$$

For a fixed k_p value we can examine some of the preliminary results of the algorithm. Let $k_p = 2.25$ then the real, non-negative, distinct, and finite zeros of $q(\omega) = q_1(\omega) + k_p q_2(\omega)$ with odd multiplicity are

$$
\begin{aligned}
\omega_0 &= 0.000000 \\
\omega_1 &= 0.426476 \\
\omega_2 &= 1.149770 \\
\omega_3 &= 4.656726 \ .
\end{aligned}
$$

The set F^* is

$$\Big\{(1, -1, 1, -1)\Big\}$$

which produces the stabilizing set of k_i values as

$$k_i \in (8.975590, 26.200133) \ .$$

By sweeping over all k_p in the valid range results in a stabilizing region depicted in Figure 2.3 where the * indicates the center of mass of the 2D region as a method of choosing a "non-fragile" controller. This controller was found to be

$$k_p = 5.919930$$
$$k_i = 17.86692 \ .$$

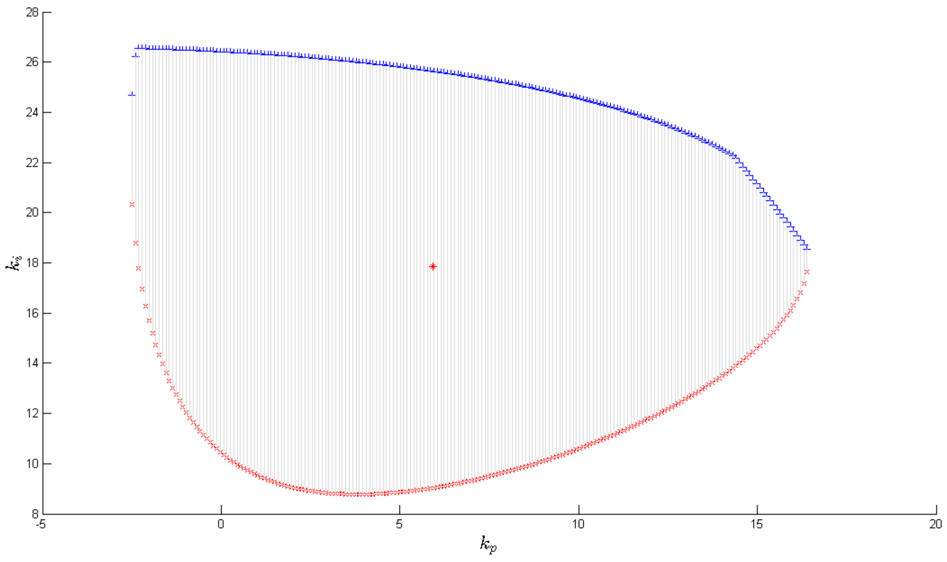

Figure 2.3: The stabilizing set of (k_p, k_i) values with "non-fragile" controller indicated by the star.

Even though verifying the entire stabilizing set of (k_p, k_i) pairs would be impossible, we will consider a representative example to this solutions validity by considering the step response of the system with the controller given by the center of mass calculation on the region and simulate the system using the MATLAB SIMULINK model of Figure 2.4.

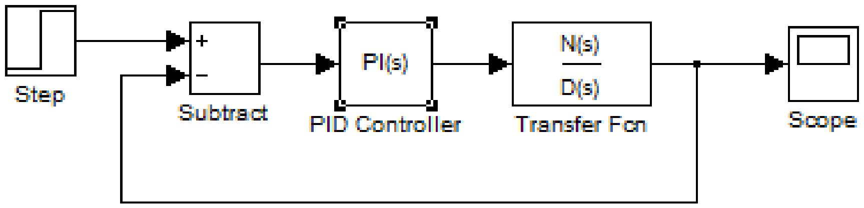

Figure 2.4: Simulation model for PI-control of an LTI system.

The (k_p, k_i) pair is $k_p = 5.919930$ and $k_i = 17.86692$. The output response is shown in Figure 2.5.

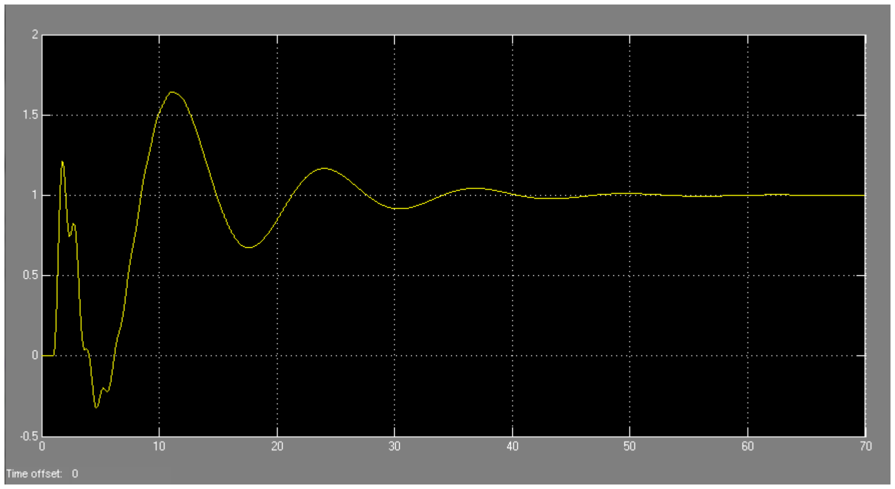

Figure 2.5: System response with $k_p = 5.919930$ and $k_i = 17.86692$.

As a second example, we consider the plant defined by

$$N(s) = s^4 + 4s^3 + 23s^2 + 46s - 12$$

$$D(s) = s^5 + 2s^4 + 23s^3 + 44s^2 + 97s + 98 \ .$$

When executing the [PI] algorithm for this specific plant on our embedded system we find the following results:

$$p_1(\omega) = -10.0\omega^{10} + 38.0\omega^8 - 346.0\omega^6 - 461.0\omega^4 + 5672.0\omega^2$$
$$p_2(\omega) = 1.0\omega^8 - 30.0\omega^6 + 137.0\omega^4 + 2668.0\omega^2 + 144.0$$
$$q_1(\omega) = -2.0\omega^9 + 48.0\omega^7 - 360.0\omega^5 + 2736.0\omega^3 - 1176.0\omega$$
$$q_2(\omega) = 1.0\omega^9 - 30.0\omega^7 + 137.0\omega^5 + 2668.0\omega^3 + 144.0\omega \ .$$

Performing the K_p narrowing routine on this example produces the finite K_p sweeping interval to be all

$$k_p \in (-1.063467, 8.166667) \ .$$

By sweeping over all k_p in the valid range results in a stabilizing region depicted in Figure 2.6 where the * indicates the center of mass of the 2D region as a method of choosing a "non-fragile" controller. In this example we have disjoint sets. Thus, two center of masses are calculated, one for each set. These two controller was found to be

$$\begin{array}{ll} k_p = -0.318000 & k_p = 3.566319 \\ k_i = -1.384091 & k_i = -0.556366 \end{array}$$

with the best "non-fragile" controller given as

$$k_p = 3.566319$$
$$k_i = -0.556366 \ .$$

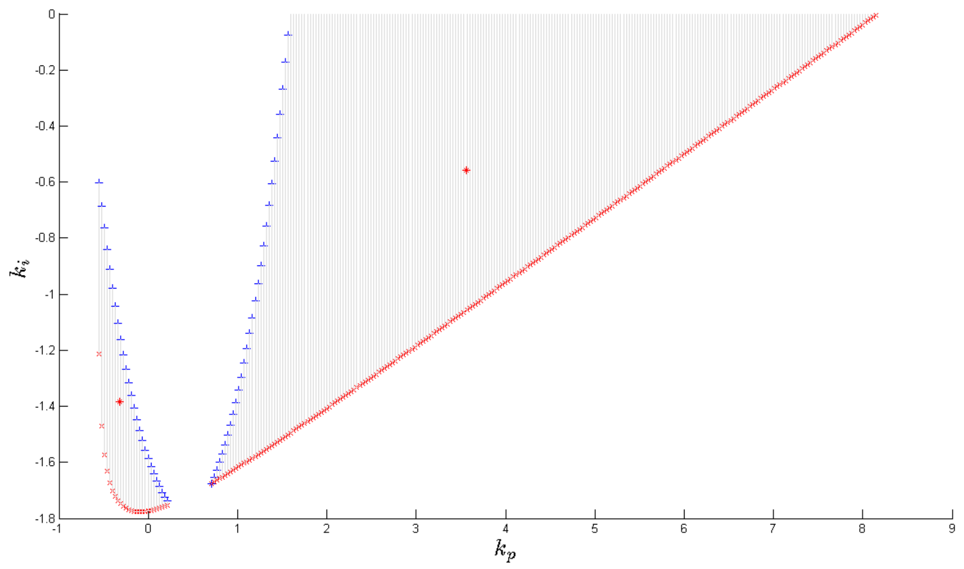

Figure 2.6: The stabilizing set of (k_p, k_i) values with "non-fragile" controller indicated by the star.

The stability of the system over all possible (k_p, k_i) pairs is impossible to verify, but as a representative example to the systems stability and correctness of the algorithm we consider the step response of the system with the controller given by the center of mass calculation on the region and simulate the system using the MATLAB SIMULINK model of Figure 2.4. The (k_p, k_i) pair is $k_p = 3.566319$ and $k_i = -0.556366$. The output response is presented in Figure 2.7.

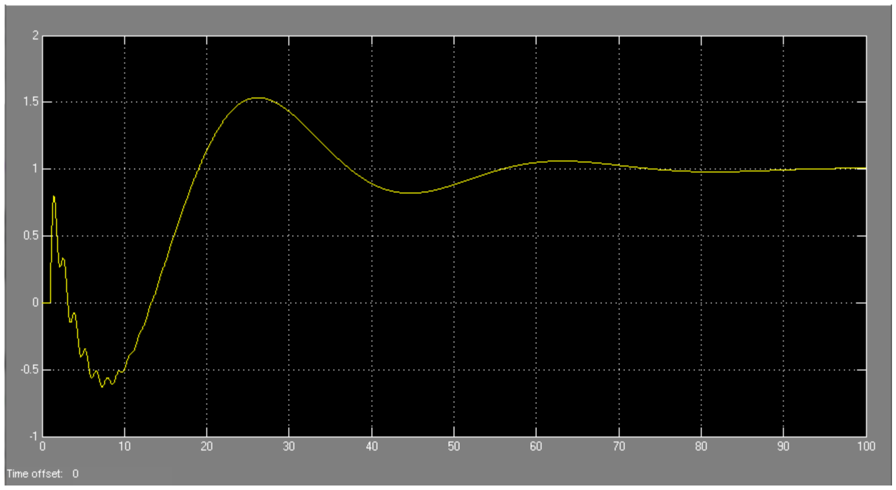

Figure 2.7: System response with $k_p = 3.566319$ and $k_i = -0.556366$.

The results of these examples verify correctness of the [PI] algorithm implemented on the embedded system because they match the results found in [24].

2.5.3 Stabilization Using a PID Controller

Let the plant be defined by

$$N(s) = s^3 - 4s^2 + s + 2$$
$$D(s) = s^5 + 8s^4 + 32s^3 + 46s^2 + 46s + 17 \; .$$

When executing the [PID] algorithm for this specific plant on our embedded system we find the following results:

$$p_1(\omega) = -12.0\omega^8 + 180.0\omega^6 - 183.0\omega^4 - 75.0\omega^2$$
$$p_2(\omega) = 1.0\omega^6 + 14.0\omega^4 + 17.0\omega^2 + 4.0$$
$$q_1(\omega) = -1.0\omega^9 + 65.0\omega^7 - 246.0\omega^5 + 22.0\omega^3 + 34.0\omega$$
$$q_2(\omega) = 1.0\omega^7 + 14.0\omega^5 + 17.0\omega^3 + 4.0\omega \; .$$

Performing the K_p narrowing routine on this example produces the finite K_p sweeping interval to be

$$k_p \in (-8.500000, 4.233366)\ .$$

For a fixed k_p value we can examine some of the preliminary results of the algorithm. Let $k_p = 1.0$ then the real, non-negative, distinct, and finite zeros of $q(\omega) = q_1(\omega) + k_p q_2(\omega)$ with odd multiplicity are

$$\begin{aligned}
\omega_0 &= 0.000000 \\
\omega_1 &= 0.742303 \\
\omega_2 &= 1.865901 \\
\omega_3 &= 7.892111\ .
\end{aligned}$$

The set F^* is

$$\Big\{(1, -1, 1, -1)\Big\}$$

which produces this set of linear inequalities

$$\begin{aligned}
-4.000000k_i + 0.000000k_d &< 0.000000 \\
2.467671k_i + -1.359721k_d &< 9.418357 \\
-0.322129k_i + 1.121521k_d &< 3.927345 \\
0.002327k_i + -0.144974k_d &< 1.080087
\end{aligned}$$

which matches exactly the results in [24] when the inequalities are normalized by the k_i coefficients. These inequalities define a convex set whose vertex points in the

(k_i, k_d) plane are

$$v_0 = (-0.000000, -6.926686)$$
$$v_1 = (-0.000000, 3.501802)$$
$$v_2 = (6.826662, 5.462592) \ .$$

Figure 2.8 illustrates this region.

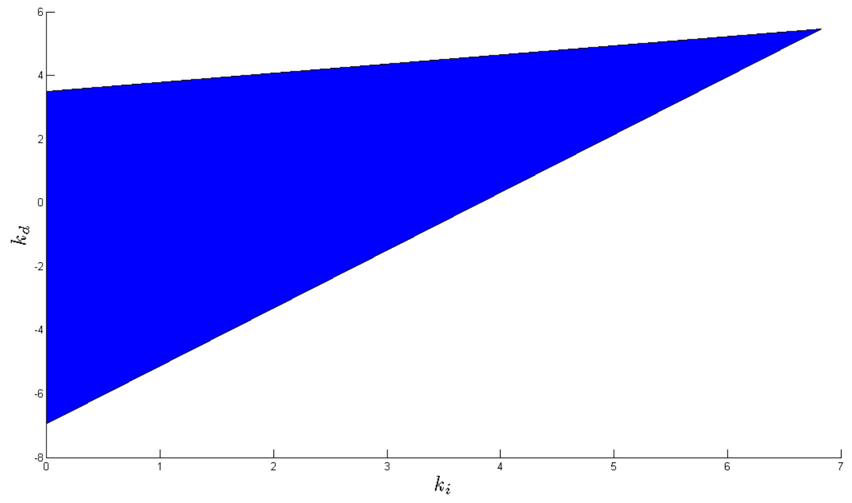

Figure 2.8: The stabilizing set of (k_i, k_d) values for $k_p = 1.0$.

Sweeping over all k_p in the valid range results in the stabilizing region depicted in Figure 2.9. Using the method of choosing an optimal "non-fragile" controller by means of the largest inscribed circle over all k_p values we obtain

$$k_p = 0.986358$$
$$k_i = 2.247527$$
$$k_d = 1.810374$$

with a radius $r = 2.247527$.

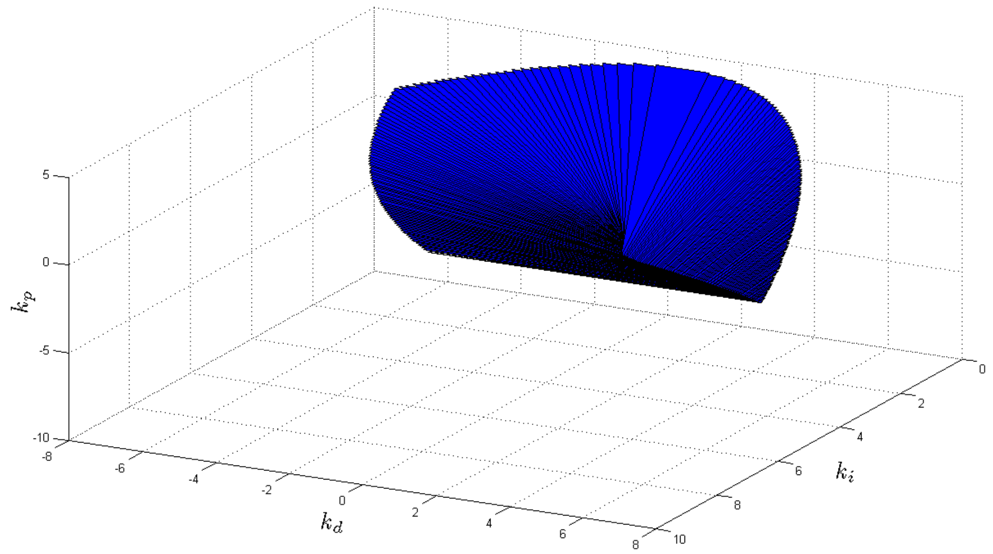

Figure 2.9: The stabilizing set of (k_p, k_i, k_d) values.

It is impossible to verify the stability of the system over all possible (k_p, k_i, k_d) triples, but as a representative example to the systems stability and correctness of the algorithm we consider the step response of the system with the controller given by the largest inscribed circle calculation on the region and simulate the system using the MATLAB SIMULINK model of Figure 2.10.

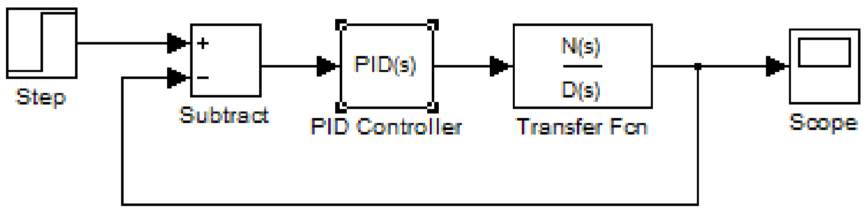

Figure 2.10: Simulation model for PID-control of an LTI system.

The (k_p, k_i, k_d) triple is $k_p = 0.986358$, $k_i = 2.247527$, and $k_d = 1.810374$. The output response is shown in Figure 2.11.

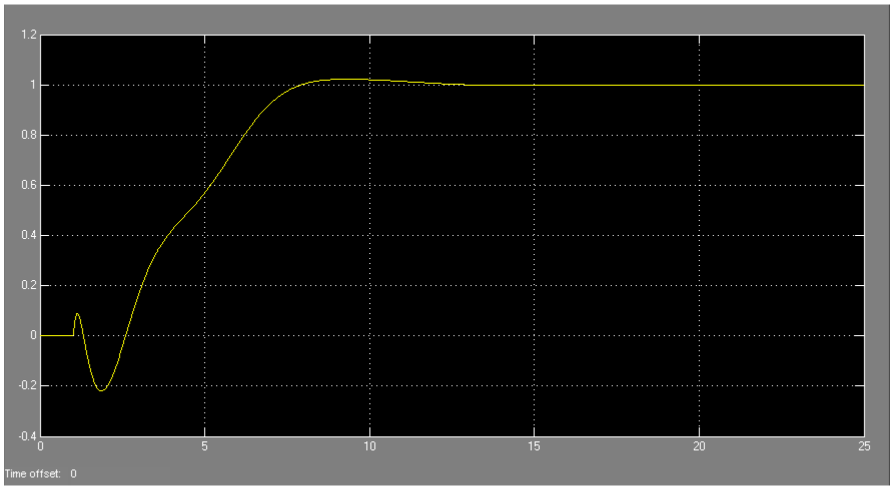

Figure 2.11: System response for $k_p = 0.98636$, $k_i = 2.24753$, and $k_d = 1.81037$.

For the next example we use the plant defined by

$$N(s) = s^3 + 3s^2 + s + 8$$
$$D(s) = s^4 + 2s^3 + 3s^2 + 7s + 14\ .$$

When executing the [PID] algorithm for this specific plant on our embedded system we get the following results.

$$p_1(\omega) = -1.0\omega^8 - 2.0\omega^6 + 20.0\omega^4 - 42.0\omega^2$$
$$p_2(\omega) = 1.0\omega^6 + 7.0\omega^4 - 47.0\omega^2 + 64.0$$
$$q_1(\omega) = -1.0\omega^7 + 8.0\omega^5 - 59.0\omega^3 + 112.0\omega$$
$$q_2(\omega) = 1.0\omega^7 + 7.0\omega^5 - 47.0\omega^3 + 64.0\omega\ .$$

Performing the K_p narrowing routine for this example produces the finite K_p sweeping interval to be

$$k_p \in (-3.272120, -1.750000) \cup (0.521717, 1.550635) \; .$$

For a fixed k_p value we can examine some of the preliminary results of the algorithm. Let $k_p = -2.0$ then the real, non-negative, distinct, and finite zeros of $q(\omega) = q_1(\omega) + k_p q_2(\omega)$ with odd multiplicity are

$$\begin{aligned} \omega_0 &= 0.000000 \\ \omega_1 &= 0.717006 \\ \omega_2 &= 1.483394 \; . \end{aligned}$$

The set F^* is

$$\Big\{ (-1, 1, -1, 1) \Big\}$$

which produces this set of linear inequalities:

$$\begin{aligned} 64.000000k_i - 0.000000k_d &< 0.000000 \\ -7.958008k_i + 4.091188k_d &< -3.167678 \\ 0.048869k_i - 0.107535k_d &< 0.384429 \\ 0.000000k_i + 1.000000k_d &< -1.000000 \; . \end{aligned}$$

These inequalities define a convex set whose vertex points in the (k_i, k_d) plane are

$$v_0 = (-0.000000, -3.574933)$$

$$v_1 = (-0.000000, -1.000000)$$

$$v_2 = (-1.878749, -4.428733)$$

$$v_3 = (-0.116048, -1.000000)\ .$$

Figure 2.12 illustrates this region.

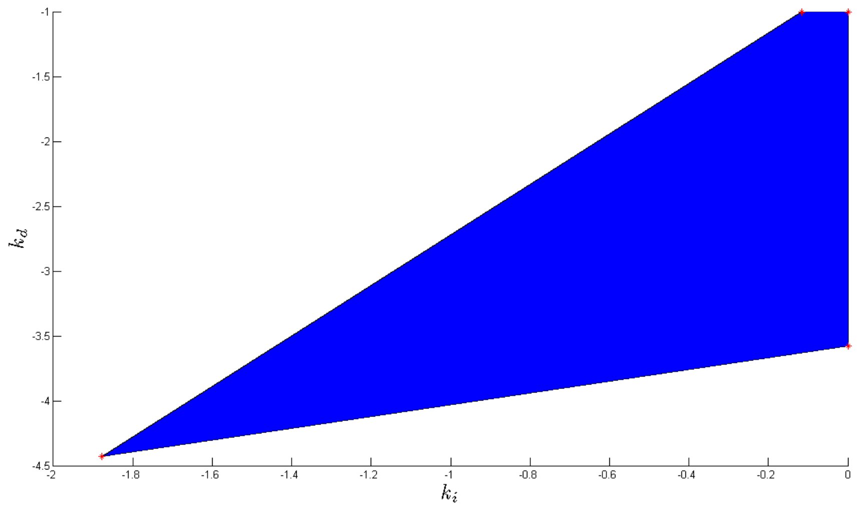

Figure 2.12: The stabilizing set of (k_i, k_d) values for $k_p = -2.0$.

By sweeping over all k_p in the valid range results in a stabilizing region depicted in Figure 2.13. Using the method of choosing an optimal "non-fragile" controller by means of the largest inscribed circle over all k_p values we obtain

$$k_p = -2.442565$$
$$k_i = -0.799320$$
$$k_d = -2.905589$$

with a radius $r = 0.799320$.

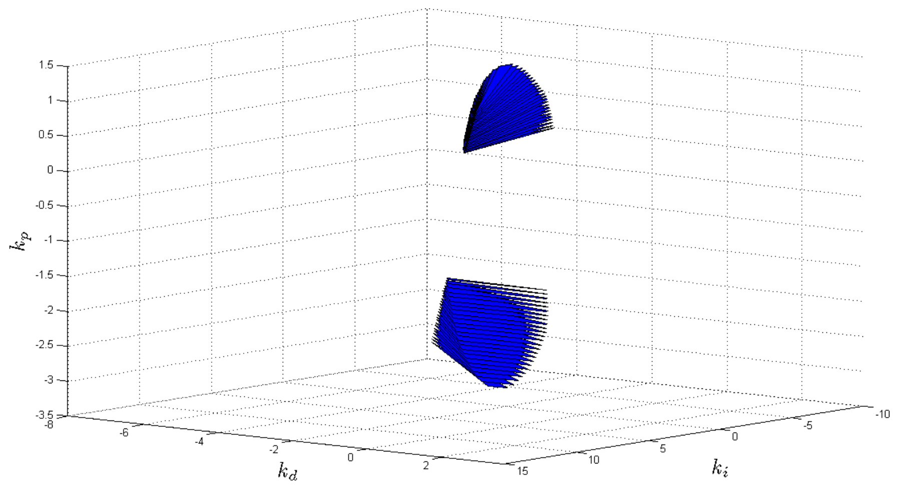

Figure 2.13: The stabilizing set of (k_p, k_i, k_d) values.

We will consider a single point inside the stability region as a representative example to the systems stability and correctness of the algorithm because verification of the stability of the system over all possible (k_p, k_i, k_d) triples would be impossible. We consider the step response of the system with the controller given by the largest inscribed circle calculation over the entire region of Figure 2.13 and simulate the system using the MATLAB command `step`. The (k_p, k_i, k_d) triple is $k_p = -2.442565$, $k_i = -0.799320$, and $k_d = -2.905589$. The output response is shown in Figure 2.14.

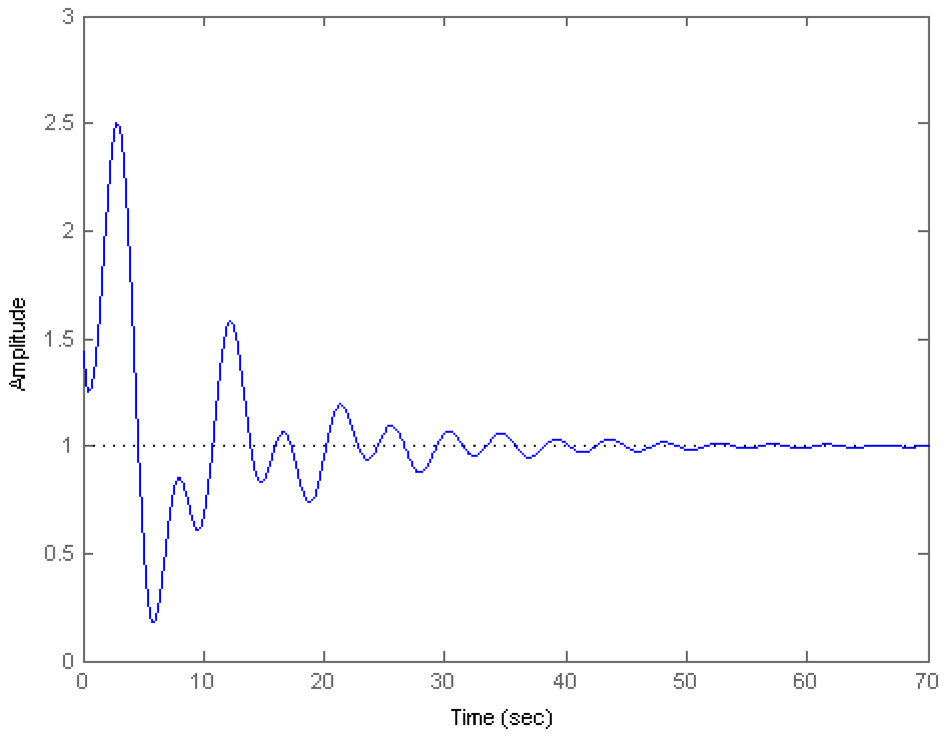

Figure 2.14: System response with $k_p = -2.442565$, $k_i = -0.799320$, and $k_d = -2.905589$.

Remark: The controller triples that have a k_p value of $k_p \in (-3.272120, -1.750000)$ are stable only via a pole/zero cancellation. Therefore, they could not be simulated via the MATLAB SIMULINK model of Figure 2.10 because the pole of the closed loop characteristic function that exists in the right half plane is fully expressed in the MATLAB SIMULINK environment even though mathematically it will cancel with a zero in the numerator. The step response was generated only after a perfect pole/zero cancelation was done. Although these controllers are theoretically stable, they would not be practically stable.

The next example illustrates the effect of replacing the pure derivative term in the PID controller with the term $\frac{k_d s}{1+Ts}$. As we saw previously, the [PID] algorithm is still applicable when a simple linear transformation is applied on the (k_p', k_i', k_d') set. Consider the plant defined by

$$N(s) = s^3 + 3s^2 + s + 8$$
$$D(s) = s^4 + 2s^3 + 3s^2 + 7s + 14$$

with $T = 0.1$. The algorithm computes the stabilizing region of (k_p', k_i', k_d') values where

$$k_p' = k_p - k_i T$$
$$k_i' = k_i$$
$$k_d' = k_d + k_p T$$

and the controller is $C'(s) = k_p' + \frac{k_i'}{s} + k_d' s$ with the plant model now represented as

$$\frac{N(s)}{(1 + Ts)D(s)} .$$

The stability region is shown in Figure 2.15. After application of the linear transformation, the stabilizing region of (k_p, k_i, k_d) values is illustrated in Figure 2.16.

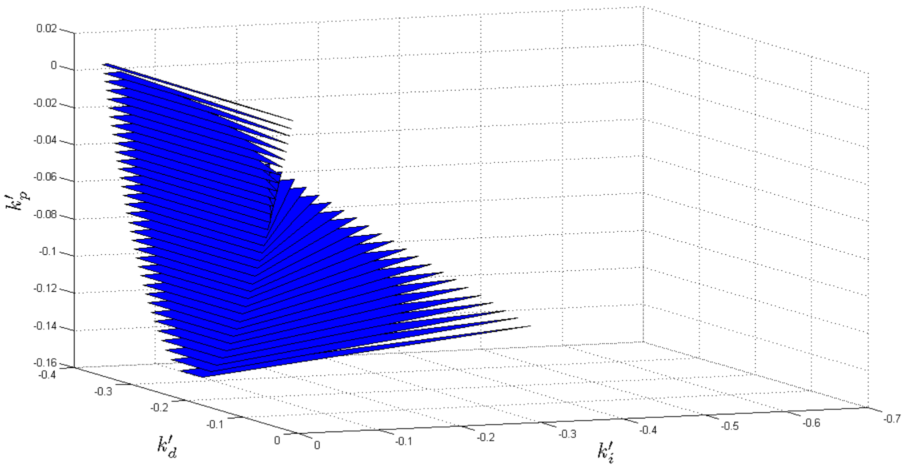

Figure 2.15: The stabilizing set of (k'_p, k'_i, k'_d) values.

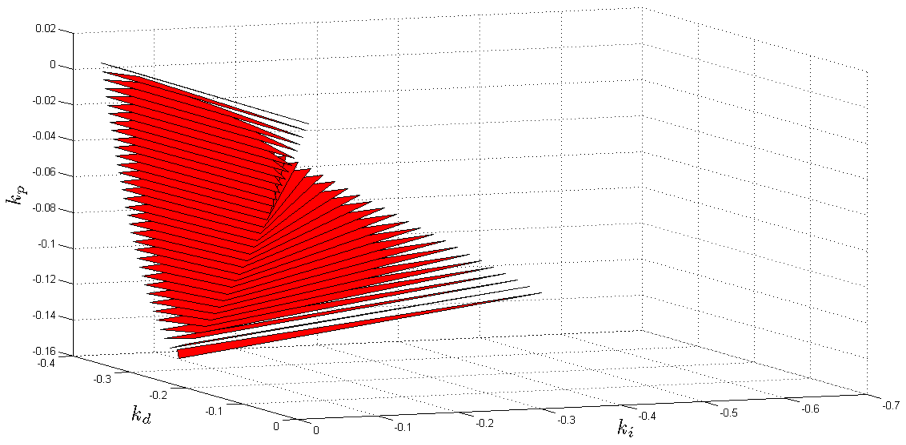

Figure 2.16: The stabilizing set of (k_p, k_i, k_d) values.

As a final example for verification of the implementation of the [PID] algorithm on our embedded system platform we consider the following plant defined by

$$N(s) = 1$$

$$D(s) = (s+1)^8 = s^8 + 8s^7 + 28s^6 + 56s^5 + 70s^4 + 56s^3 + 28s^2 + 8s + 1 \ .$$

The [PID] algorithm calculates the stability region illustrated in Figure 2.17.

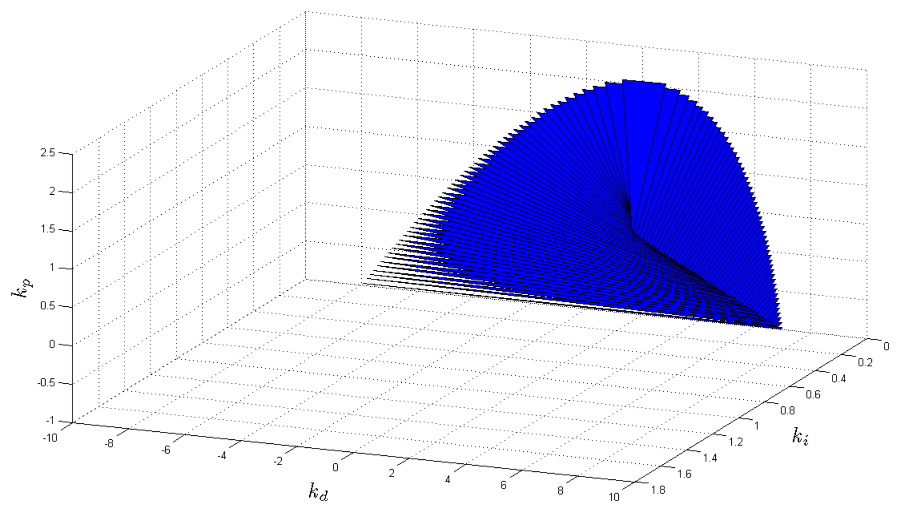

Figure 2.17: The stabilizing set of (k_p, k_i, k_d) values.

The optimal "non-fragile" control determined via the largest inscribed circle metric, was found to exist at

$$k_p = 1.321673$$
$$k_i = 0.425626$$
$$k_d = 5.162580$$

with a radius of $r = 0.425626$. This search was performed over the narrowed K_p set of

$$k_p \in (-1.000000, 2.075064)\ .$$

If we examine the specific value of $k_p = 1.321673$ we produce the stability region of Figure 2.18 with the largest inscribed circle noted in the figure by the red circle.

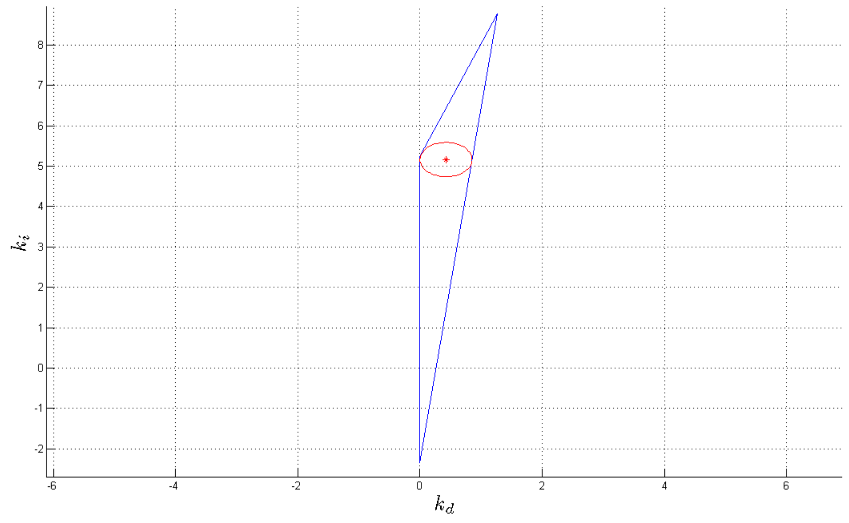

Figure 2.18: The stabilizing (k_i, k_d) region for $k_p = 1.32167$ with largest inscribed circle.

A slight discrepancy exists between the results of [24] and the results produced here. In [24] we are provided with the following optimal "non-fragile" controller

$$k_p = 1.32759$$

$$k_i = 0.42563$$

$$k_d = 5.15291$$

with a radius of $r = 0.42563$. This is due to the method of quantizing the K_p interval. Our method undoubtedly varies from that in [24]; however, if we force the examination of $k_p = 1.32759$ into our embedded system implementation, we find the exact values as above

$$k_p = 1.327590$$
$$k_i = 0.425632$$
$$k_d = 5.152907$$

with a radius of $r = 0.425632$. Ultimately, the more finely quantized K_p is, the more accurate the final controller will become.

The stability of the system over all possible (k_p, k_i, k_d) triples is impossible to verify, but as a representative example to the systems stability and correctness of the algorithm we consider the step response of the system with the controller given by the largest inscribed circle calculation on the region and simulate the system using the MATLAB SIMULINK model of Figure 2.10. The (k_p, k_i, k_d) triple is $k_p = 1.321673$, $k_i = 0.425626$, and $k_d = 5.162580$. The output response is presented in Figure 2.19.

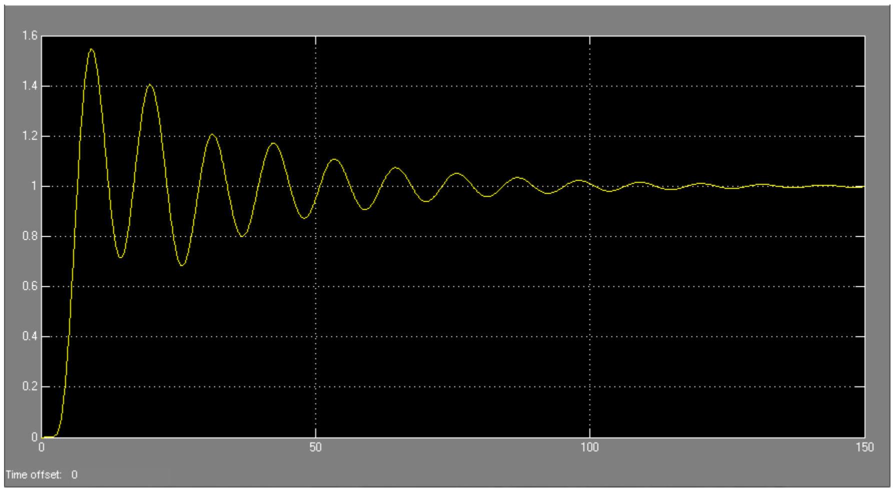

Figure 2.19: System response for $k_p = 1.32167$, $k_i = 0.42563$, and $k_d = 5.16258$.

These results verify correctness of the [PID] algorithm implemented on the embedded system because they match the results found in [24].

2.6 Summary

In this chapter we presented a computational formulation of the constant gain, PI, and PID algorithms presented in [24] for use on embedded systems platforms. A `Polynomial` Class was developed along with many sub algorithms to efficiently compute the results. All the algorithms were verified against examples. In the next chapter we will utilize all the algorithms presented here for use in optimal PID controller design under a specific set of optimality criteria with the goal of real time optimal auto-tuning of PID controllers.

3. PID CONTROLLER DESIGN UNDER CONSTRAINED OPTIMALITY CRITERIA

One of the major applications of the algorithms defined in the previous chapter is the ability to optimally design a PID controller to optimize a given performance index. Once the entire set of stabilizing (k_p, k_i, k_d) values have been found, each candidate controller can be analyzed for specific performance metrics and the optimal one chosen. Performance metrics include the H_2 and H_∞ norm of the closed loop transfer function and such time domain characteristics as settling time, overshoot, and undershoot. Having this ability within an embedded system can be a powerful tool for optimally designing an auto-tuning controller that will always remain near optimal operation.

3.1 The Optimality Criterion

For the purposes of this implementation we examined five performance criteria to achieve an optimal controller design

1. Minimum H_2 norm of the closed loop transfer function
2. Minimum H_∞ norm of the closed loop transfer function
3. Minimum Settling Time
4. Minimum Overshoot
5. Minimum Undershoot

Thus, given the numerator and denominator polynomials of a plant model we are able to run the [P], [PI], or [PID] algorithms discussed in chapter 2 and obtain the entire set of stabilizing k_p, (k_p, k_i), or (k_p, k_i, k_d) values, respectively. Using this entire set we can construct all possible controllers that will stabilize the plant. Then, by using all possible controllers we can evaluate each of these performance criteria and choose the controller that meets our specific need. Recall that in chapter 2 we included a quantization of the feasibility region as part of the [PID] algorithm. This quantization makes the data of all possible controllers available so that we may utilize this data to find optimal controllers. While the algorithm is computing the linear inequalities that define the convex sets, it is easier at that time to produce the feasibility region and provide a quantization of that region for use in evaluation of different controller's performance indices.

3.1.1 H_2 Optimal Control

In H_2 optimal control we seek a controller that minimizes the $\mathcal{L}_2$ norm of the error signal. The $\mathcal{L}_2$ space is a Hilbert space [37] of matrix-valued (or scalar-valued) functions and consists of all complex matrix functions F such that

$$\int_{-\infty}^{\infty} Trace[F^*(j\omega)F(j\omega)]d\omega < \infty$$

The inner product on this Hilbert space is defined as

$$\langle F, G\rangle = \frac{1}{2\pi}\int_{-\infty}^{\infty} Trace[F^*(j\omega)G(j\omega)]d\omega$$

for $F, G \in \mathcal{L}_2$, and the inner product induced norm being $\|F\|_2 = \langle F, F\rangle^{\frac{1}{2}}$. However, it is sometimes easier to compute such norms from state space models. This characterization introduces the H_2 space defined as a closed subspace of $\mathcal{L}_2$ with matrix functions $F(s)$ analytic in $Re(s) > 0$. The induced norm of H_2 is defined as [37]

$$\begin{aligned} \|F\|_2^2 &= \sup_{\sigma>0}\{\frac{1}{2\pi}\int_{-\infty}^{\infty} Trace[F^*(\sigma+j\omega)F(\sigma+j\omega)]d\omega\} \\ &= \frac{1}{2\pi}\int_{-\infty}^{\infty} Trace[F^*(j\omega)F(j\omega)]d\omega \end{aligned}$$

As shown the norms for both H_2 and $\mathcal{L}_2$ are the same. For ease of computation we consider a state space form of the H_2 norm. Considering the stable continuous time system with real transfer function $G(s)$ and corresponding state space matrices A, B, C, and D, we have

$$\|G\|_2^2 = trace(B^TXB)$$

where X is obtained from the following Lyaponov equation

$$AX + XA^T + BB^T = 0.$$

Using Eigen Value Decomposition(EVD) the solution to X can be easily found via the method found in [38]. In order to perform EVD within the TiDSP C++ environment, we chose the open source library called *ALGLIB* [39]. This library has no external dependencies and is modular in its design allowing only the code base necessary to implement EVD, leaving out all the other functionality of ALGLIB to deliver the smallest possible code size; an important quality for embedded systems that generally lack a large program memory.

3.1.2 H_∞ Optimal Control

In H_∞ optimal control we seek a controller that minimizes the $\mathcal{L}_\infty$ norm of the error signal. The $\mathcal{L}_\infty$ space is a Banach space [37] of matrix-valued (or scalar-valued) functions F that are bounded on the imaginary axis with norm

$$\|F\|_\infty = ess \sup_{\omega \epsilon R} \bar{\sigma}[F(j\omega)].$$

where $\bar{\sigma}$ is the maximum singular value of the matrix function $F(j\omega)$. However, it may be easier to compute such norms from state space models. This characterization introduces the H_∞ space defined as a closed subspace of $\mathcal{L}_\infty$ with matrix functions $F(s)$ analytic in $Re(s) > 0$ and bounded on the imaginary axis. The induced norm of H_2 is defined as [37]

$$\|F\|_\infty = \sup_{Re(s)>0} \bar{\sigma}[F(s)] = ess \sup_{\omega \epsilon R} \bar{\sigma}[F(j\omega)].$$

As shown, the norms for both H_∞ and $\mathcal{L}_\infty$ are the same. The problem of H_∞ optimal control is often cast into the sensitivity minimization problem in which we seek a controller that will cause the weighted sensitivity function or closed loop error transfer function to be small in the H_∞ norm [40]. Thus, we seek to find the $\min \|W(s)T(s)\|_\infty$ where $W(s)$ is the weighting function [24] defined[1] as $W(s) = \frac{s+0.1}{s+1}$ and $T(s) = \frac{C(s)G(s)}{1+C(s)G(s)}$ is the complimentary sensitivity function (though any suitable sensitivity function may be used), and $G(s)$ is the plant transfer function.

Computationally, this norm is easier to solve in state space form and algorithms exist in the literature: [41] and [42]. Unfortunately, no analytic formulation exists and iterative approximations must be computed and searched. The simplest method used is known as the Bisecting Algorithm [37]. The MATLAB implementation for calculating the H_∞ norm is based off the algorithm found in [41]. To implement this in the TiDSP C++ environment we used a combination of both the MATLAB

[1]This is not unique; any suitable weighting may be chosen

algorithm and the Bisection Algorithm for simplicity of implementation.

3.1.3 Settling Time, Overshoot, and Undershoot

Three common performance metrics in feedback control are the settling time, overshoot, and undershoot of the system; others include peak time and rise time. The settling time is defined as the time required for the system to settle within a certain percentage of the input amplitude when excited by a step response [1]. A typical percentage value commonly used is 2%. The overshoot is a measure of the peak amplitude of the signal during its transient response relative to a step input. This is typically measured as a percentage of the reference input value. The undershoot is a measure of the minimum amplitude of the signal relative to some baseline, typically zero. This, like the overshoot, is often measured as a percentage of the input reference value.

For second order systems, some of these performance metrics have closed form expressions based on the damping ratio ζ. For a 2% settle time, $T_s = \frac{4}{\zeta\omega_n}$ and the percent overshoot, $P.O. = 100e^{\frac{-\zeta\pi}{\sqrt{1-\zeta^2}}}$. However, no such closed form expressions exist for higher ordered systems. Thus an analytic approach would not be general enough for our purposes. The best way to consider these performance metrics for an arbitrary plant order is through simulation of the system and data analysis.

There are numerous methods of numerical integration to simulate a dynamic system. The most basic and simplest, though least accurate, is the Euler method for integration. This is a first order, fixed step, iterative integration method which allows for fast computation at the expense of larger error. Higher ordered integration methods, like Runge-Kutta, are more accurate but more complex in their computational load. Besides increasing the order of integration, one can also move to a variable step size in the integration processes. This further increases the accuracy allowing for finer resolution around critical points, reducing the overall error, but greatly increases complexity and computation time. For the purpose of simplicity the first order Euler method was chosen to simulate the system in order to determine the settling time, overshoot and undershoot data.

To obtain the settling time, overshoot and undershoot, the first step is to take the closed loop transfer function and convert it into a valid state space model. The state space form chosen was the Controllable Canonical form, but it does not matter which form is used. The Controllable Canonical form was chosen because it is easier to discretize the system in this form. Discretization of the system generates a set of first order difference equations in the time domain. Then, using a fixed step size each state of the system can be calculated for each time step over a set interval of discrete time. This is how the system is simulated and the output of the system is stored.

Using the stored output data of the simulation over a span of 100 seconds we are able to determine the settling time, overshoot and undershoot. The settling time is found by performing a scan of the data and counting the number of consecutive data points that are within 5% of the input[2]. Since the step size and end time of the simulation are known, we can back count the amount of time from the end of the simulation in order to find the exact time, the settling time, at which the final response value remained within 5% of the step input value. The overshoot is a simple search of the output data for the maximum value. The percent that this value is above the step input value is the percent overshoot. And similarly, the undershoot is the minimum value that occurs. Therefore, by using the simple Euler method for integration with a fixed step size and the basic definitions of settling time,

[2]We used a more relaxed 5% settling time parameter rather than the 2% typically seen. Additionally the input is a step with a value of 1.

overshoot, and undershoot, we are able to determine these performance metrics for each stabilizing controller found. An exhaustive search through all these performance metrics garners us the ability to find an optimal controller.

3.2 Examples and Results

We will now put the [P], [PI], and [PID] algorithms to use in obtaining optimal PID controllers based on various performance criteria. In every case we consider the plant model and weighting function

$$\begin{aligned} G(s) &= \frac{N(s)}{D(s)} = \frac{s-1}{s^2+0.8s-0.2} \\ W(s) &= \frac{s+0.1}{s+1} \ . \end{aligned}$$

The search method employed in these examples is brute force. Once the complete characterization of the stabilizing set is found via the [P], [PI] and [PID] algorithms, the region is then quantized to produce a large set of unique k_p, (k_p, k_i), or (k_p, k_i, k_d) values, respectively. Then for each value the H_∞ norm, H_2 norm, Settling Time, Overshoot, and Undershoot are calculated where applicable. The smallest value is found and the corresponding P, PI, or PID parameters is chosen as the optimal one.

3.2.1 Constant Gain Design Using H_∞ and H_2 Norms

Executing the [P] algorithm on the given plant provides us a narrowed K_p set of

$$k_p \in (-0.800000, -0.200000) \ .$$

Using a quantization of this interval at a resolution of 200 evenly spaced data points we calculate the H_∞ and H_2 Norms for each of the 200 data points. Then a complete search over the set is performed for the minimum H_∞ and H_2 Norms. The result for the case of the H_∞ norm was

$$\min_{k_p \in (-0.8, -0.2)} \|W(s)T(s)\|_\infty = 0.518470$$

at $k_p = -0.255500$ A plot of the data points is shown in Figure 3.1.

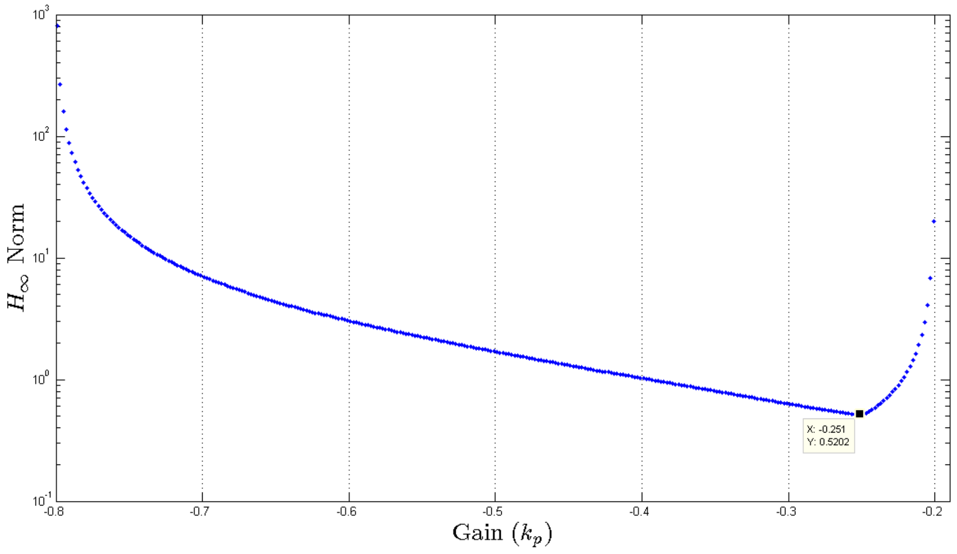

Figure 3.1: H_∞ norm of $W(s)T(s)$ for constant gain stabilization.

The result for the case of the H_2 norm was

$$\min_{k_p \in (-0.8, -0.2)} \|W(s)G(s)S(s)\|_2 = 1.038303$$

at $k_p = -0.267500$. A plot of the data points is presented in Figure 3.2.

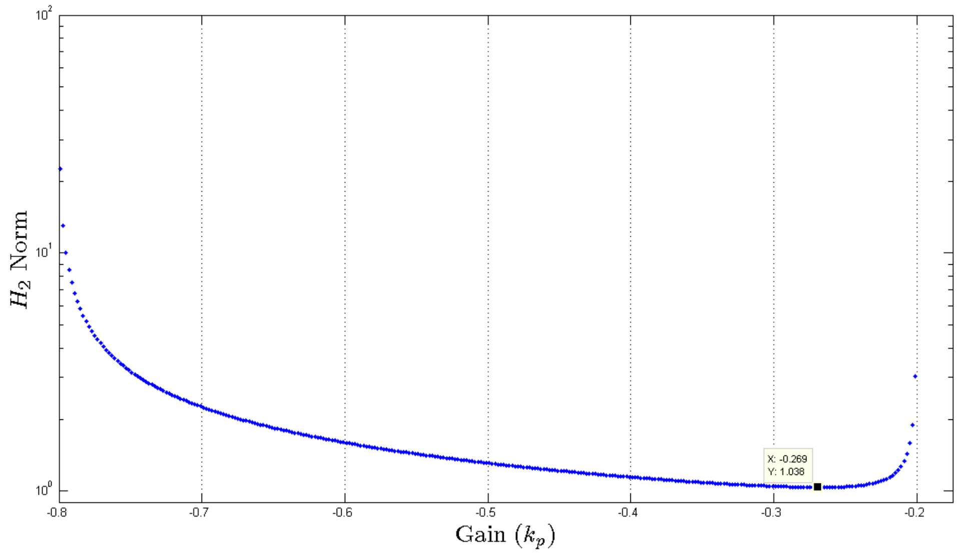

Figure 3.2: H_2 norm of $W(s)G(s)S(s)$ for constant gain stabilization.

These results verify correctness of the [P] algorithm implemented on the embedded system for use in optimal constant gain controller design because they match the results found in [24].

3.2.2 PI Controller Design Using H_∞ and H_2 Norms and Time Domain Performance Metrics

Applying the [PI] algorithm to the given plant provides us with a narrowed K_p set of

$$k_p \in (-1.800000, -0.200000) \; .$$

A full characterization of the (k_p, k_i) set of stabilizing controllers is shown in Figure 3.3 with a "non-fragile" controller indicated by the star, found via the center of mass calculation of the region.

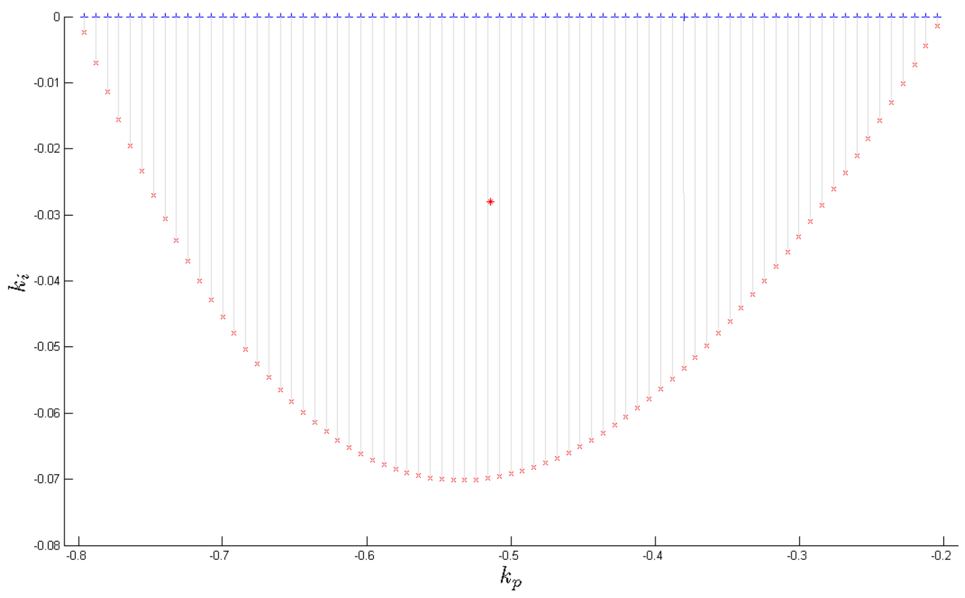

Figure 3.3: The stabilizing set of (k_p, k_i) values.

Using a quantization resolution of 300 data points for the K_p interval and 200 data points for each K_i interval corresponding to a specific k_p value, totalling 60,000 unique controllers, we calculated the H_∞ norm, H_2 norm, settling time, overshoot, and undershoot. Using these values we can search for the minimum for each performance criterion.

It may be the case [24] that an optimal condition is reached when $k_i = 0$. Mathematically this forces the PI controller into a Constant Gain controller. However, the point of using a PI controller is for asymptotic step tracking. Practically, such an implementation of $k_i = 0$ will cause the closed loop system to be destabilized. Therefore, we place a constraint on k_i so that we only consider controllers where $k_i \leq -0.01$. This choice is of course arbitrary and was chosen to more closely follow the results of [24]. Under this constraint we now proceed with the results.

For the optimal H_∞ controller we find that

$$\min_{k_p \in (-1.8,-0.2)} \|W(s)T(s)\|_\infty = 0.930681$$

and this occurs at $k_p = -0.314667$ and $k_i = -0.010023$.

For the optimal H_2 controller we find that

$$\min_{k_p \in (-1.8,-0.2)} \|W(s)G(s)S(s)\|_2 = 1.184342$$

and this occurs at $k_p = -0.320000$ and $k_i = -0.010022$.

Plots of all H_∞ and H_2 values for each (k_p, k_i) pair is shown in Figures 3.4 and 3.5, respectively.

Remark: These values do not perfectly match those found in [24]. There are two reasons for this difference. The first reason affects all the results presented here. How the region is quantized affects which controller values are evaluated. That is, the method in which we quantize the interval does not allow for our algorithm to consider the (k_p, k_i) pair found in [24]. If we force the algorithm to evaluate the optimal pair found in [24] then we obtain 0.931322 for H_∞ and 1.183423 for H_2, which in the case of H_2 is an exact match. The second reason affects the H_∞ values. Due to the computational nature of finding the H_∞ norm of the closed loop transfer function, we elected a less accurate but more computationally efficient version. This was because we are implementing the algorithms on to an embedded system and computational efficiency carries a higher importance. Despite the lower accuracy the results are still acceptable.

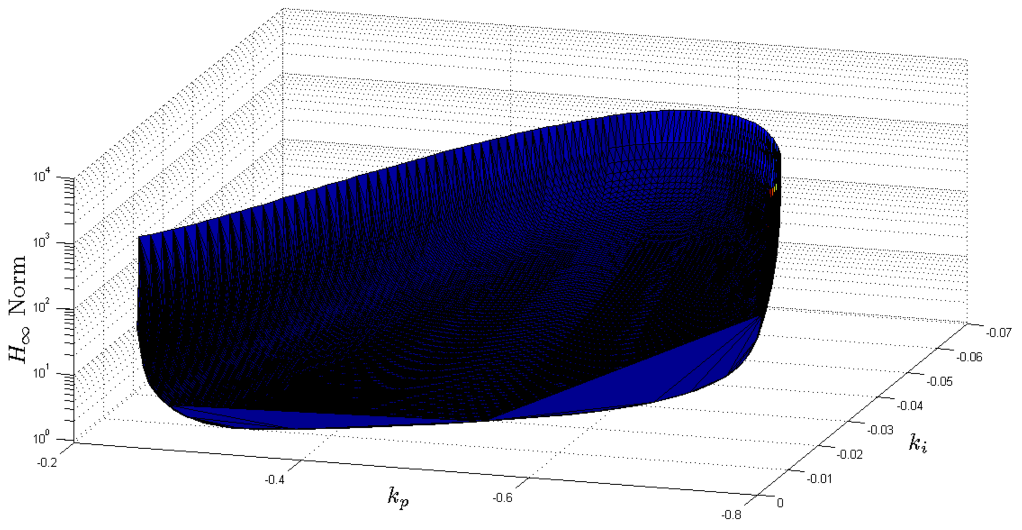

Figure 3.4: H_∞ norm of $W(s)T(s)$ vs. (k_p, k_i) for PI controllers.

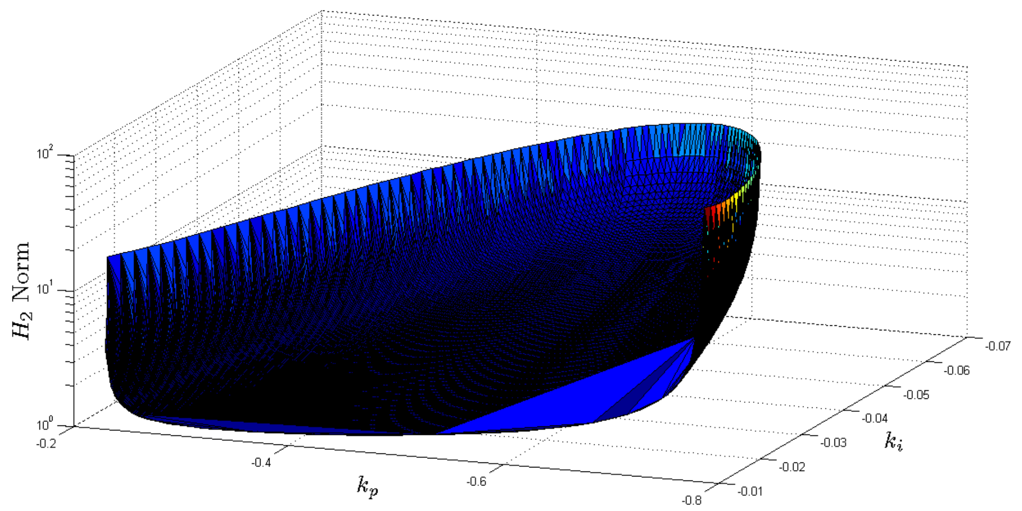

Figure 3.5: H_2 norm of $W(s)G(s)S(s)$ vs. (k_p, k_i) for PI controllers.

For settling time, overshoot, and undershoot we obtained the plots shown in Figures 3.6, 3.7, and 3.8, respectively. For each time domain characteristic we found the optimal minimum and corresponding controller parameters to be.

$$\begin{aligned}\text{Settling Time(5\%)}: &\ 21.800000, k_p = -0.346667, k_i = -0.013382\\ \text{Overshoot(\%)}: &\ 154.880593, k_p = -0.458667, k_i = -0.010038\\ \text{Undershoot(\%)}: &\ 10.152900, k_p = -0.266667, k_i = -0.010029\end{aligned}$$

where undershoot and overshoot are specified as percentages and we consider a 5% settling time in seconds.

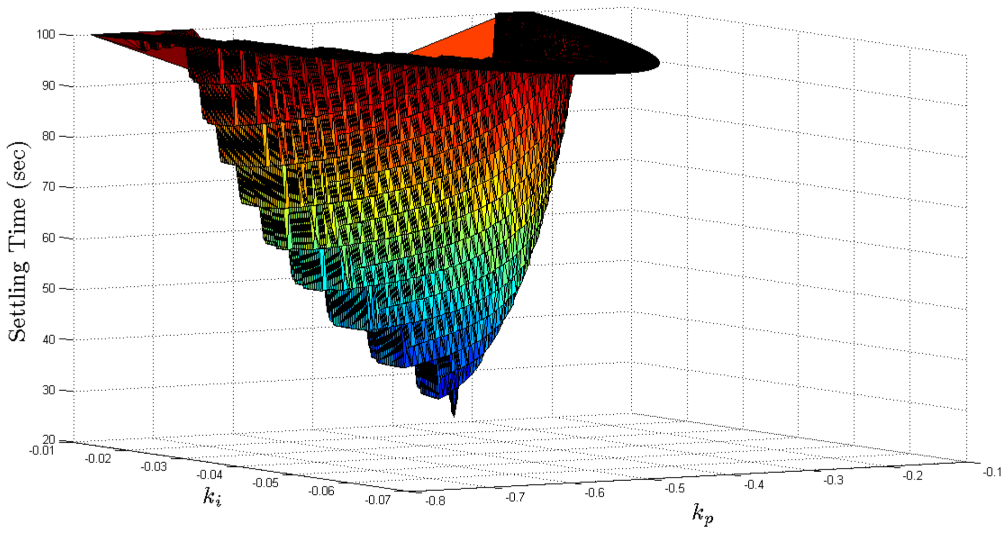

Figure 3.6: Plot of settling time vs. (k_p, k_i).

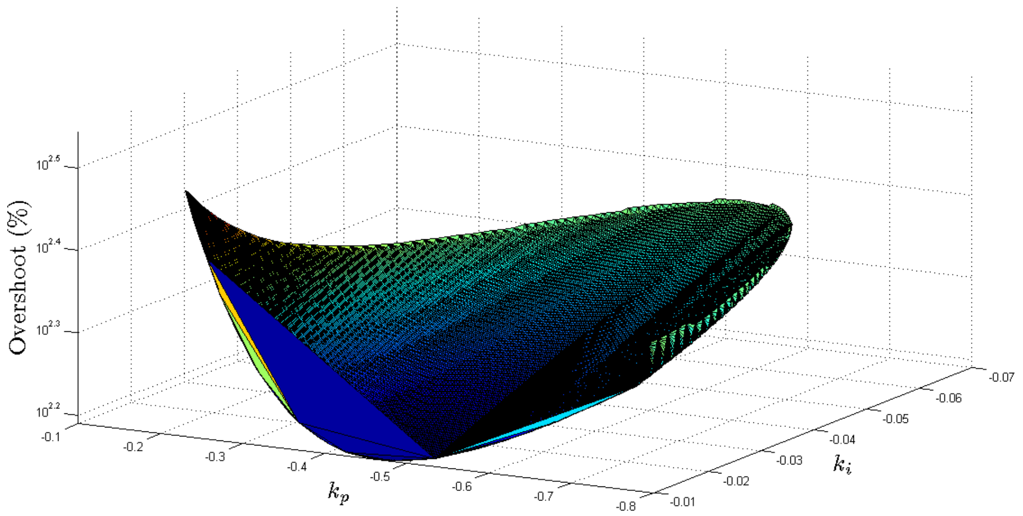

Figure 3.7: Plot of overshoot vs. (k_p, k_i).

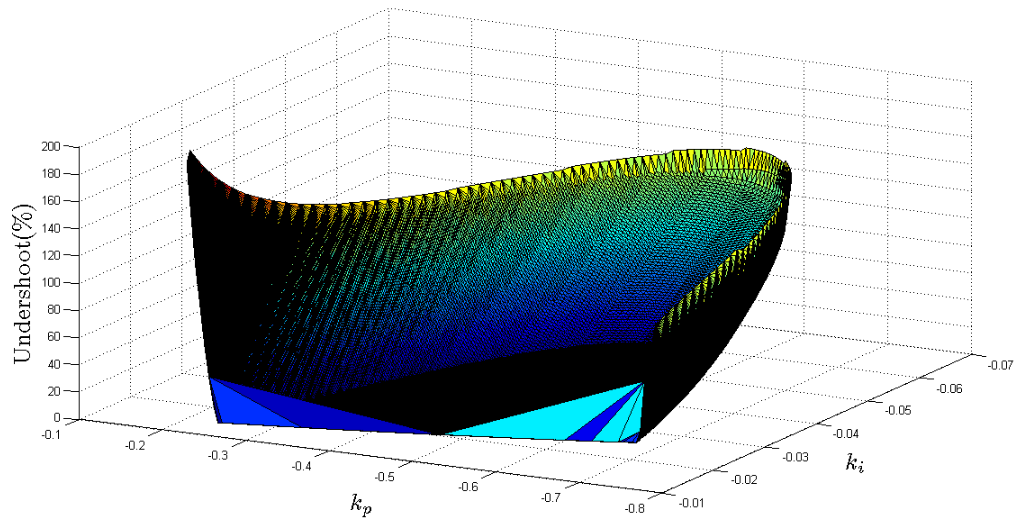

Figure 3.8: Plot of undershoot vs. (k_p, k_i).

To make full use of this algorithm we now consider using multiple performance metrics to define an optimal controller. As was done in [24] let us consider the following design criteria.

1. Settling time ≤ 30 secs.

2. Overshoot $\leq 200\%$.

Executing the algorithm and searching over the parameter space for these specific performance criteria produces a subset of valid controllers found in Table 3.1.

Table 3.1: Short list of controllers with settling time ≤ 30 and overshoot ≤ 2.

k_p	k_i	Settling Time	Overshoot	Undershoot
-0.43733	-0.02195	27.930	171.785	17.749
-0.43200	-0.02013	28.770	169.818	17.463
-0.42667	-0.02172	28.720	172.557	17.261
-0.42133	-0.02052	29.360	171.545	16.992
-0.41600	-0.02262	29.430	175.186	16.803
-0.34667	-0.01521	29.140	182.767	13.597
-0.34133	-0.01278	22.600	181.600	13.314
-0.33600	-0.01110	26.760	181.797	13.050
-0.32533	-0.01079	25.620	187.207	12.593
-0.32000	-0.01002	27.080	189.515	12.352

For validation of the algorithm applied to this example we provide the simulation results of the MATLAB SIMULINK system of Figure 2.4 using the controller parameters of $k_p = -0.43200$ and $k_i = -0.02013$. The results are presented in Figure 3.9.

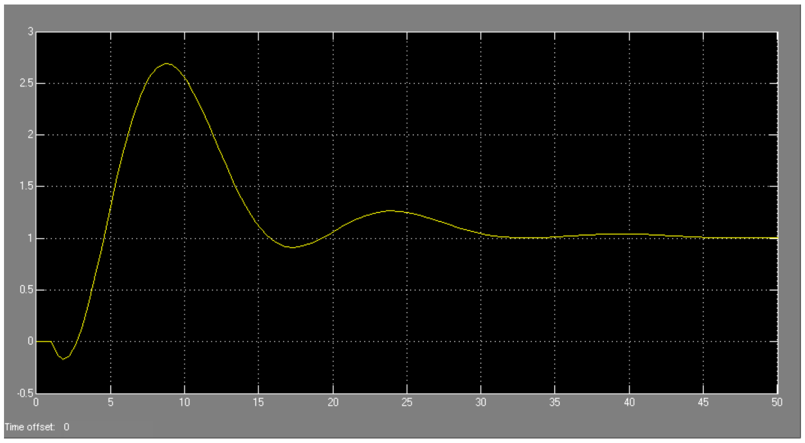

Figure 3.9: Step response for $k_p = -0.43200$ and $k_i = -0.02013$ with settling time ≤ 30 and overshoot ≤ 2.

3.2.3 PID Controller Design Using H_∞ and H_2 Norms and Time Domain Performance Metrics

Using the [PID] algorithm on the given plant provides us with a narrowed K_p set of

$$k_p \in (-1.800000, -0.200000) \ .$$

A full characterization of the (k_p, k_i, k_d) set of stabilizing controllers is shown in Figure 3.10.

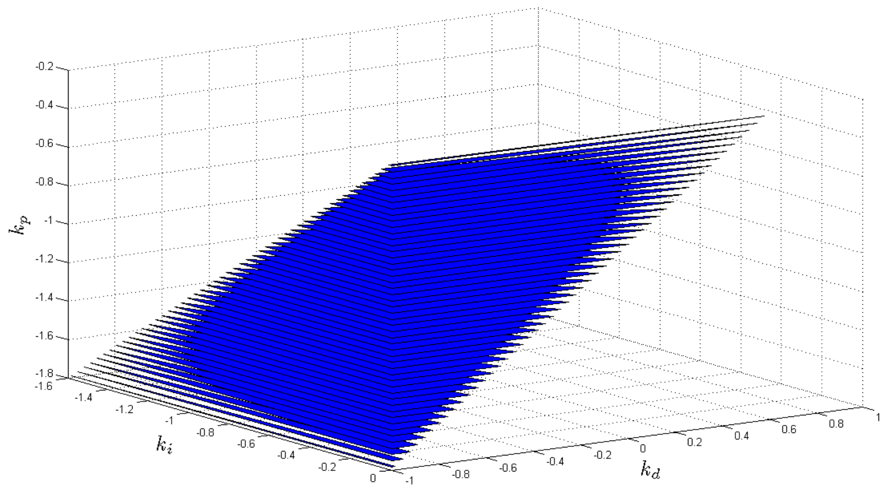

Figure 3.10: The stabilizing set of (k_p, k_i, k_d) values.

We will now consider regions of Figure 3.10 that produce the H_∞ and H_2 optimal controllers. Using a quantization resolution of 100 data points for the K_p interval and 25 data points for each K_i and K_d region corresponding to a specific k_p value, totalling 62,500 unique controllers, these values are used to find the minimum performance metrics.

As in the case of PI controllers, it may be the case [24] that an optimal condition is reached when $k_i = 0$. Thus, as before, we place a constraint on k_i so that we only consider controllers where $k_i \leq -0.01$. Under this constraint we now proceed with the computation of the results. For the optimal H_∞ controller we find that

$$\min_{k_p \in (-1.8, -0.2)} \|W(s)T(s)\|_\infty = 0.560194$$

and this occurs at $k_p = -0.368000$, $k_i = -0.010080$, and $k_d = -0.326960$. The stabilizing region corresponding to $k_p = -0.368000$ is given in Figure 3.11 and the plot of the H_∞ norm over this plane is given in Figure 3.12.

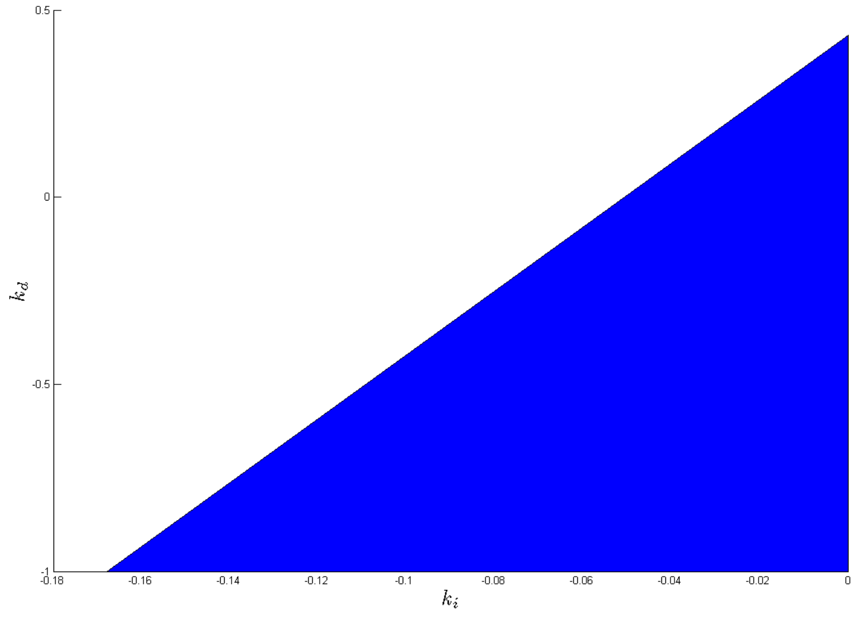

Figure 3.11: The stabilizing set of (k_i, k_d) values for $k_p = -0.368000$.

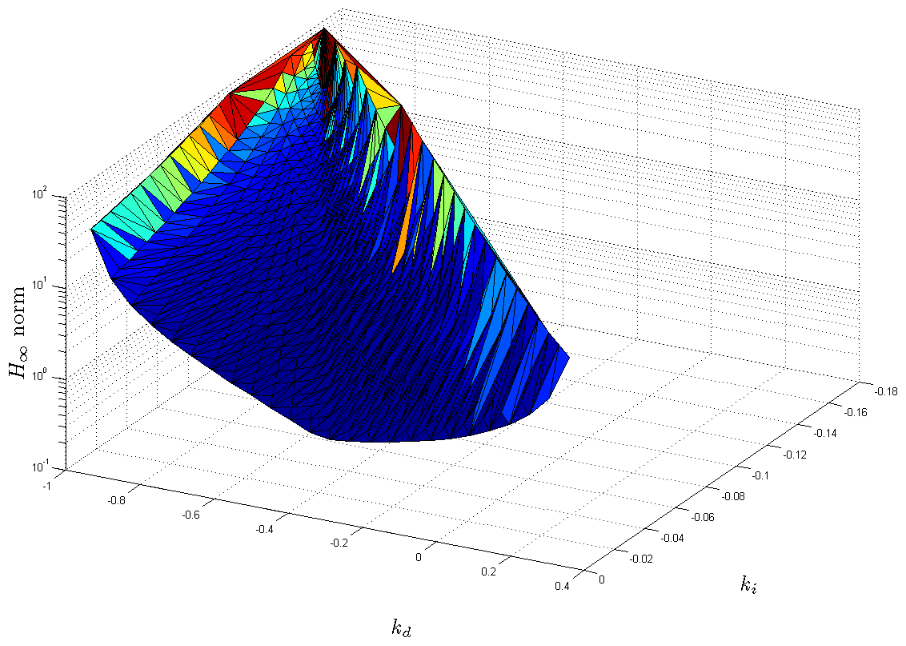

Figure 3.12: H_∞ norm of $W(s)T(s)$ for $k_p = -0.368000$.

For the optimal H_2 controller we find that

$$\min_{k_p \in (-1.8, -0.2)} \|W(s)G(s)S(s)\|_2 = 1.031678$$

and this occurs at $k_p = -0.304000$, $k_i = -0.010400$, and $k_d = -0.326960$. The stabilizing region corresponding to $k_p = -0.304000$ is given in Figure 3.13 and the plot of the H_2 norm over this plane is given in Figure 3.14.

Remark: These values do not perfectly match those found in [24]. The reasons are the same as noted in the section of this chapter relating to the PI controller results.

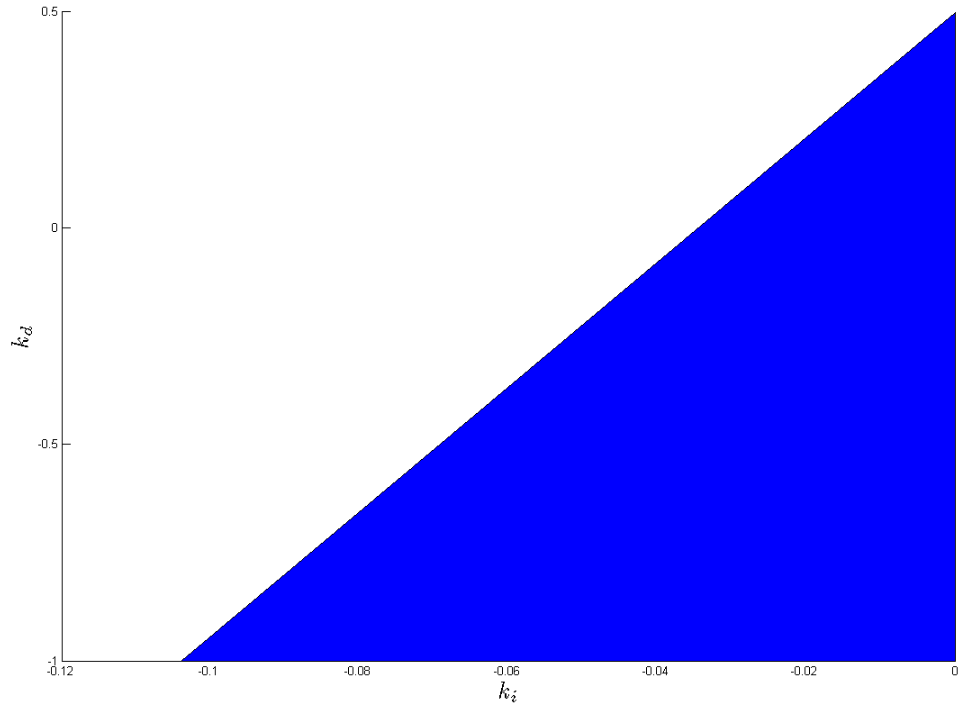

Figure 3.13: The stabilizing set of (k_i, k_d) values for $k_p = -0.304000$.

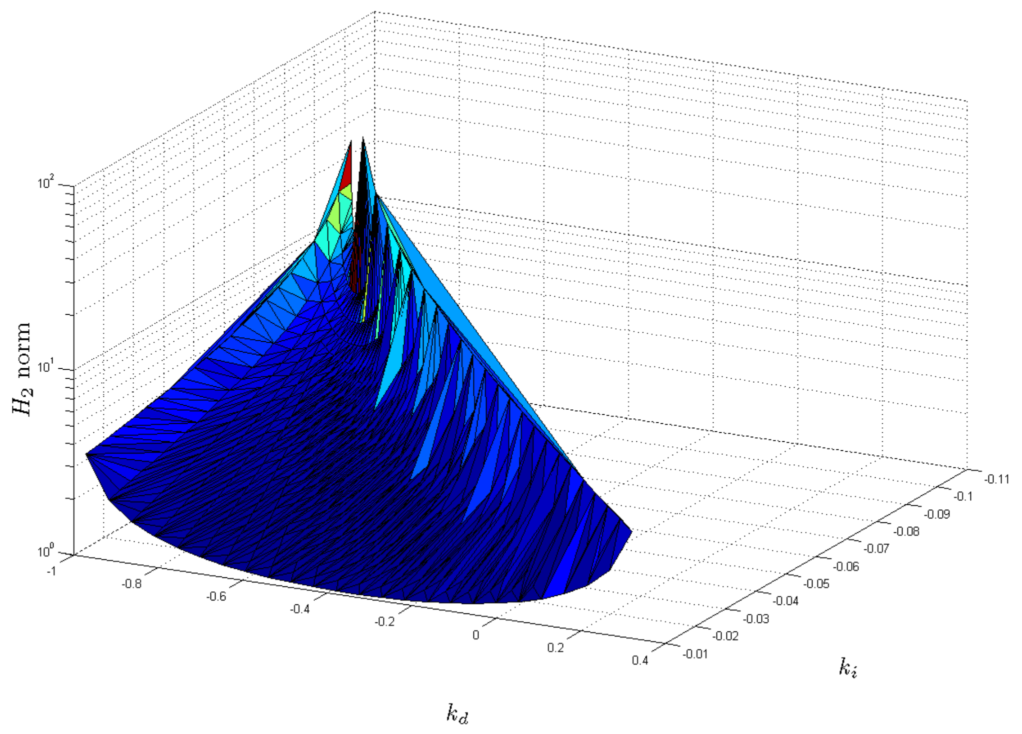

Figure 3.14: H_2 norm of $W(s)G(s)S(s)$ vs. (k_i, k_d) for $k_p = -0.304000$.

Searching over the entire (k_p, k_i, k_d) space of Figure 3.10 we found the minimum settling time, overshoot and undershoot to be

$$\text{Settling Time(5\%)}: \ 0.240000, k_p = -1.776000, k_i = -0.788000, k_d = -0.999760$$

$$\text{Overshoot(\%)}: \ 27.371123, k_p = -0.976000, k_i = -0.015520, k_d = -0.983850$$

$$\text{Undershoot(\%)}: \ 10.455415, k_p = -0.272000, k_i = -0.010080, k_d = 0.024982$$

Plots of each (k_i, k_d) plane for the respective k_p values can be found for settling time in Figure 3.15, overshoot in Figure 3.17, and undershoot in Figure 3.19. Additionally the settling time, overshoot, and undershoot plot over each plane are plotted in Figures 3.16, 3.18, 3.20, respectively.

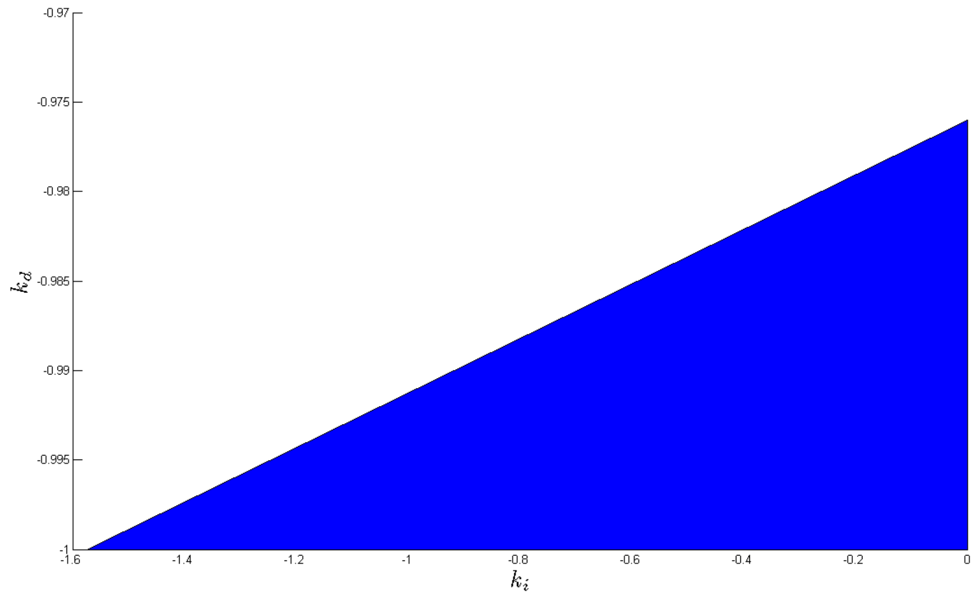

Figure 3.15: The stabilizing set of (k_i, k_d) values for $k_p = -1.776000$.

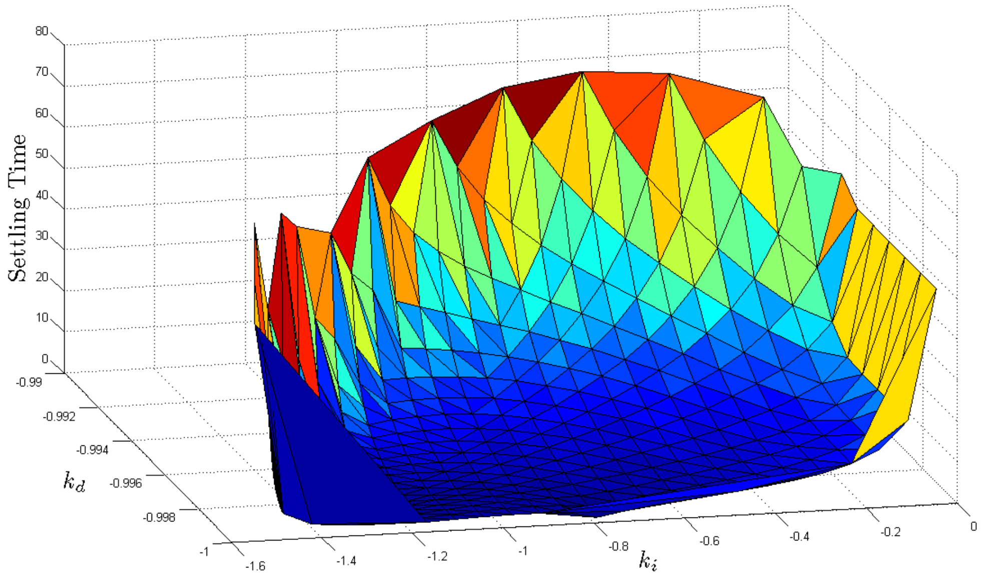

Figure 3.16: Settling time vs. (k_i, k_d) values for $k_p = -1.776000$.

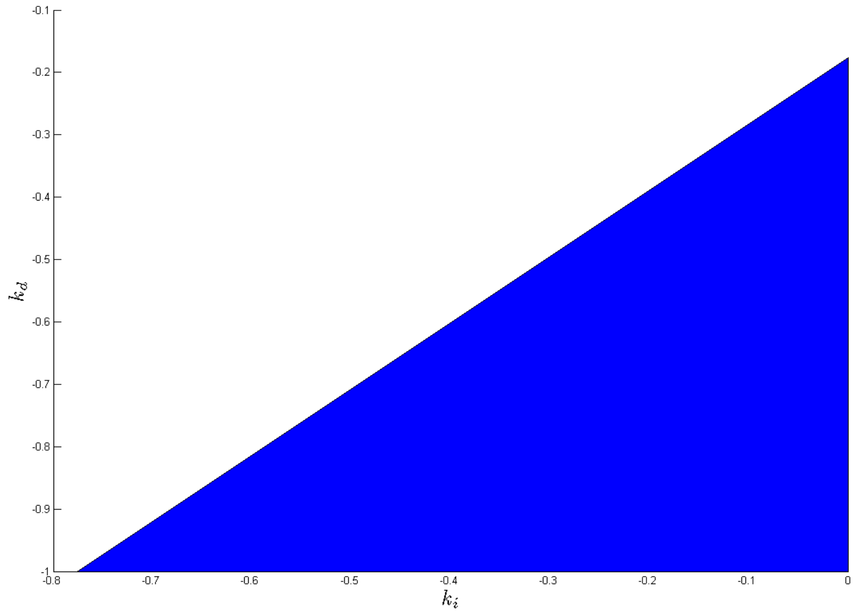

Figure 3.17: The stabilizing set of (k_i, k_d) values for $k_p = -0.976000$.

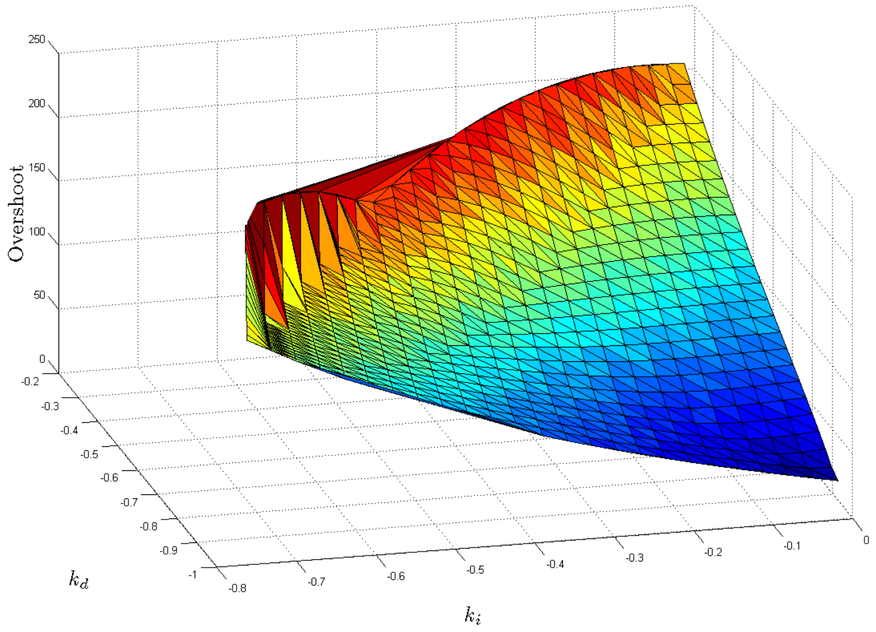

Figure 3.18: Overshoot vs. (k_i, k_d) values for $k_p = -0.976000$.

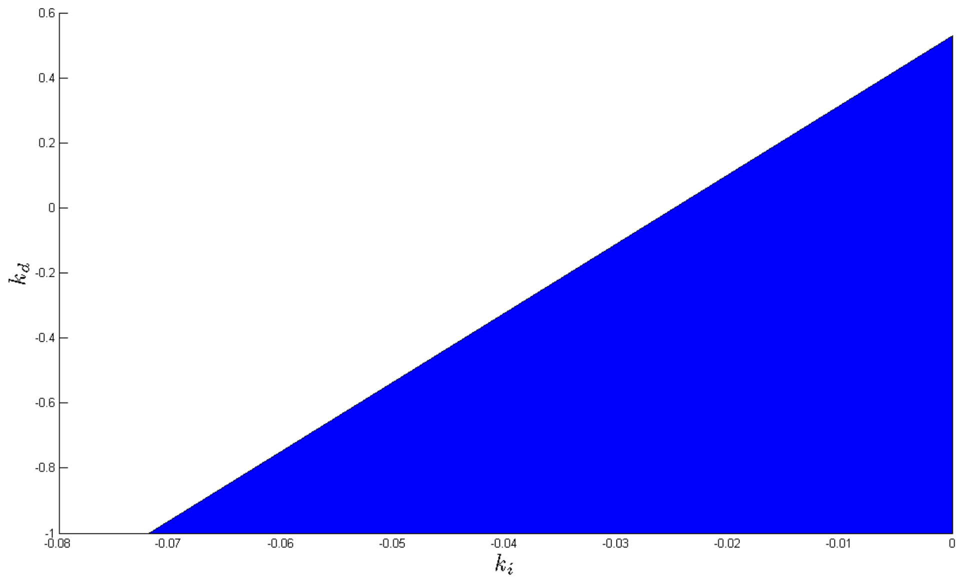

Figure 3.19: The stabilizing set of (k_i, k_d) values for $k_p = -0.272000$.

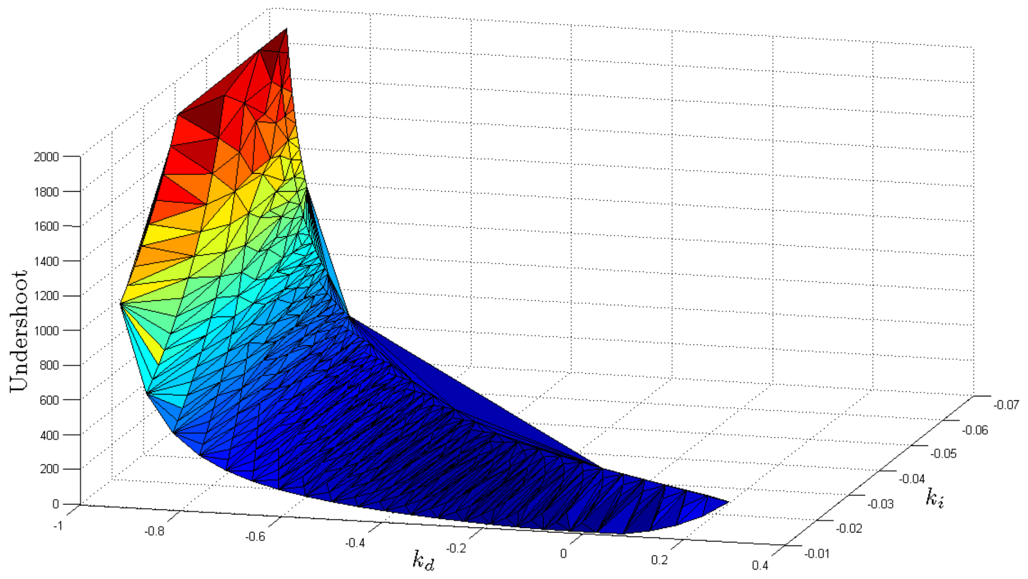

Figure 3.20: Undershoot vs. (k_i, k_d) values for $k_p = -0.272000$.

To make full use of this algorithm we now consider using multiple performance metrics to define an optimal controller. As was done in [24] let us consider the following design criteria.

1. Settling time ≤ 10 secs.
2. Undershoot $\leq 200\%$.
3. $|e_{ss_{ramp}}| \leq 1$, where $e_{ss_{ramp}}$ is the steady state error to a unit ramp input.

Executing the algorithm and searching over the parameter space for these specific performance criteria produces the set of valid controllers found in Table 3.2.

Table 3.2: Controllers with settling time ≤ 10, undershoot ≤ 2 and $k_i \leq -0.2$.

k_p	k_i	k_d	Settling Time	Overshoot	Undershoot
-1.0080	-0.2101	-0.6601	9.4700	151.7345	194.1813
-0.9920	-0.2059	-0.6532	9.6800	148.1025	188.3560
-0.9920	-0.2059	-0.6293	9.9300	156.8952	169.7523
-0.9760	-0.2018	-0.6463	9.8800	144.6588	182.7568
-0.9120	-0.2136	-0.6643	8.6600	126.7059	197.9170
-0.8960	-0.2088	-0.6583	8.9600	124.0649	192.6441
-0.8800	-0.2040	-0.6522	9.2700	121.5755	187.5546

To validate the algorithm applied to this example we provide the simulation results of the MATLAB SIMULINK system of Figure 2.10 using the controller parameters, $k_p = -0.8800$, $k_i = -0.2040$, $k_d = -0.6522$. The results are given in Figure 3.21.

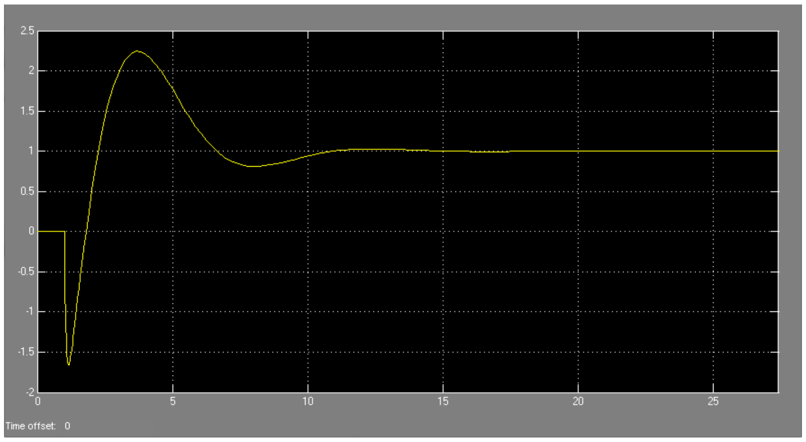

Figure 3.21: Step response for $k_p = -0.8800$, $k_i = -0.2040$, and $k_d = -0.6522$ with settling time ≤ 10, undershoot ≤ 2 and $k_i \leq -0.2$.

The algorithm is also well equipped to handle disjoint stabilizing sets. Consider the following example where

$$G(s) = \frac{N(s)}{D(s)} = \frac{s^3 + 3s^2 + 9}{s^4 + 2s^3 + 3s^2 + 7s + 14}$$

and using the same weighting function, $W(s)$, as before. Executing the [PID] algorithm on the given plant provides us with a narrowed K_p set of

$$k_p \in (-1.870784, -1.555556) \cup (0.315687, 0.533263)\ .$$

A full characterization of the (k_p, k_i, k_d) set of stabilizing controllers is given in Figure 3.22.

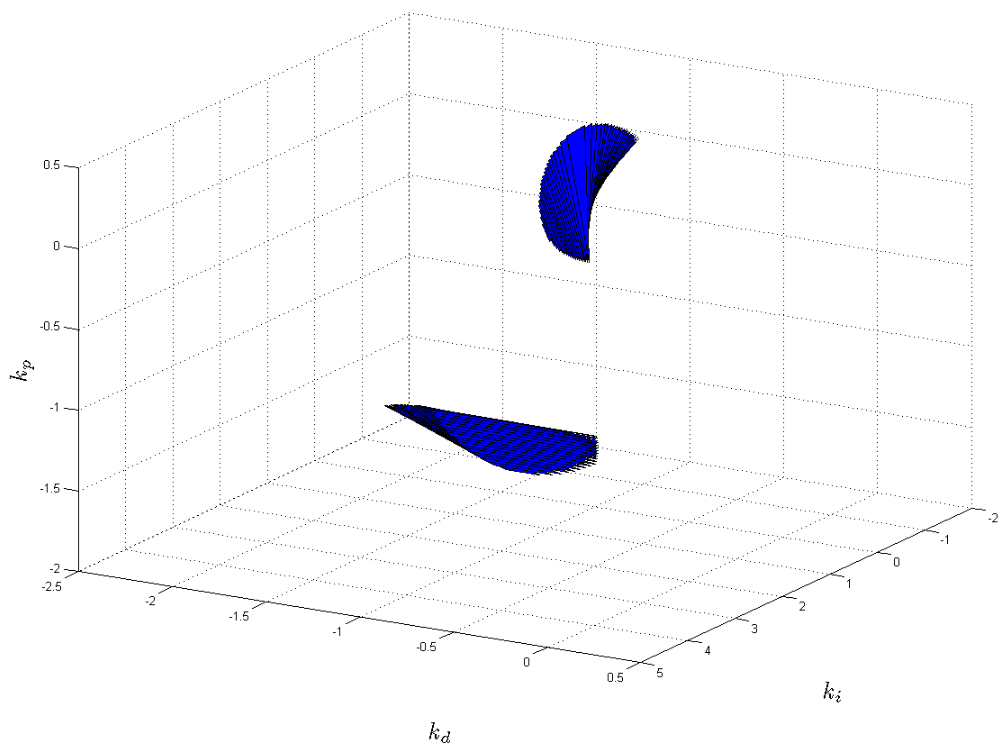

Figure 3.22: The stabilizing set of (k_p, k_i, k_d) values.

We will now consider regions of Figure 3.22 that produce the H_∞ and H_2 optimal controllers. Using a quantization resolution of 100 data points for each K_p interval and 25 data points for each K_i and K_d region corresponding to a specific k_p value, totalling 125,000 unique controller, these values are used to calculate the minimum performance metrics.

For the same reason explained previously, we place a constraint on k_i so that we only consider controllers where $|k_i| \geq 0.01$. Under this constraint we now proceed with the results. For the optimal H_∞ controller we calculate that

$$\min_{k_p \in (-1.87,-1.56) \cup (0.32,0.53)} \|W(s)T(s)\|_\infty = 6.669036$$

and this occurs at $k_p = -1.610720$, $k_i = -0.018520$, and $k_d = -1.391080$. The stabilizing region corresponding to $k_p = -1.610720$ is given in Figure 3.23 and the plot of the H_∞ norm over this plane is given in Figure 3.24.

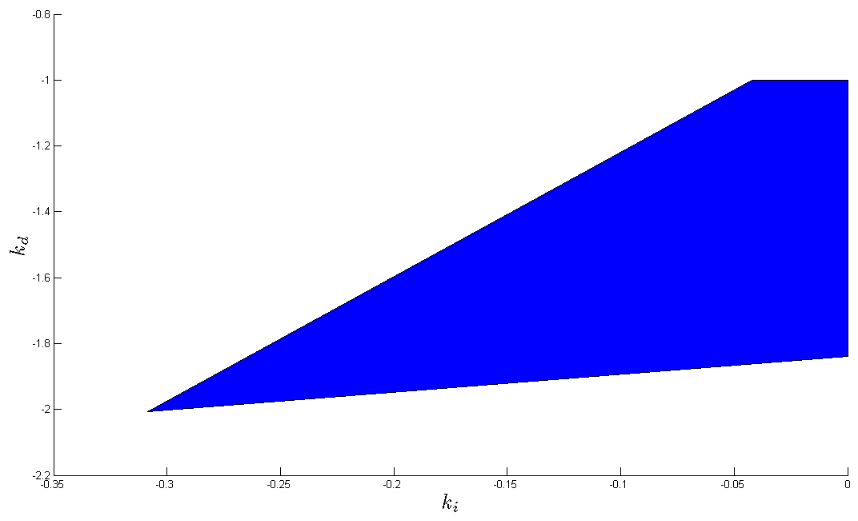

Figure 3.23: The stabilizing set of (k_i, k_d) values for $k_p = -1.610720$.

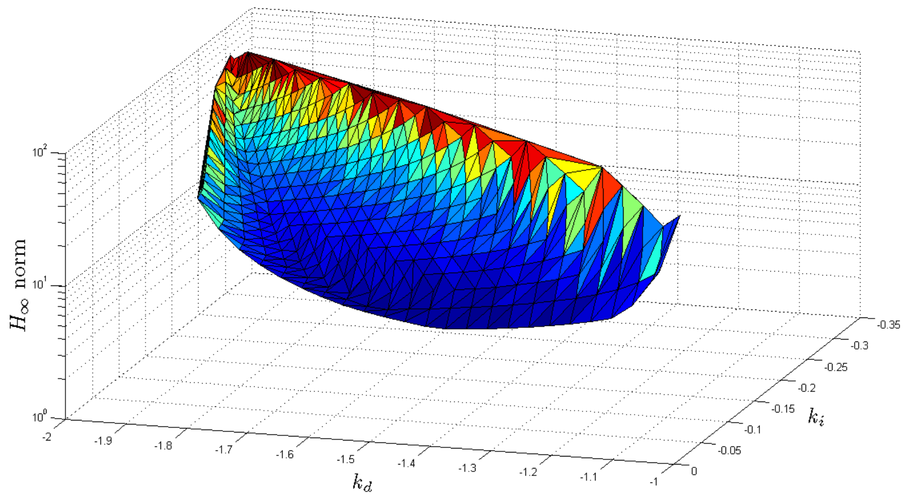

Figure 3.24: H_∞ norm of $W(s)T(s)$ for $k_p = -1.610720$.

For the optimal H_2 controller we calculate that

$$\min_{k_p \in (-1.87,-1.56)\cup(0.32,0.53)} \|W(s)G(s)S(s)\|_2 = 1.374129$$

and this occurs at $k_p = -1.648548$, $k_i = -0.010104$, and $k_d = -1.391080$. The stabilizing region corresponding to $k_p = -1.648548$ is given in Figure 3.25 and the plot of the H_2 norm over this plane is given in Figure 3.26.

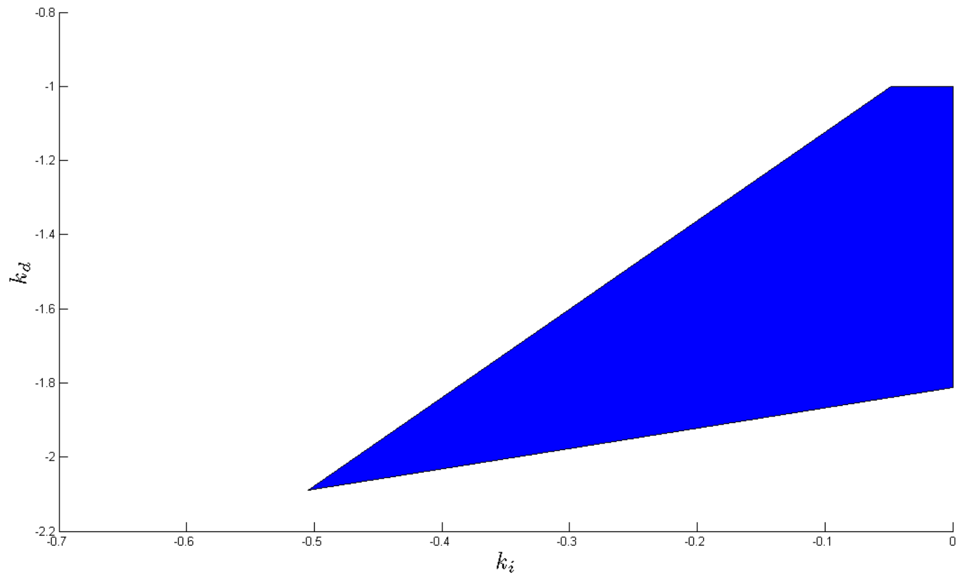

Figure 3.25: The stabilizing set of (k_i, k_d) values for $k_p = -1.648548$.

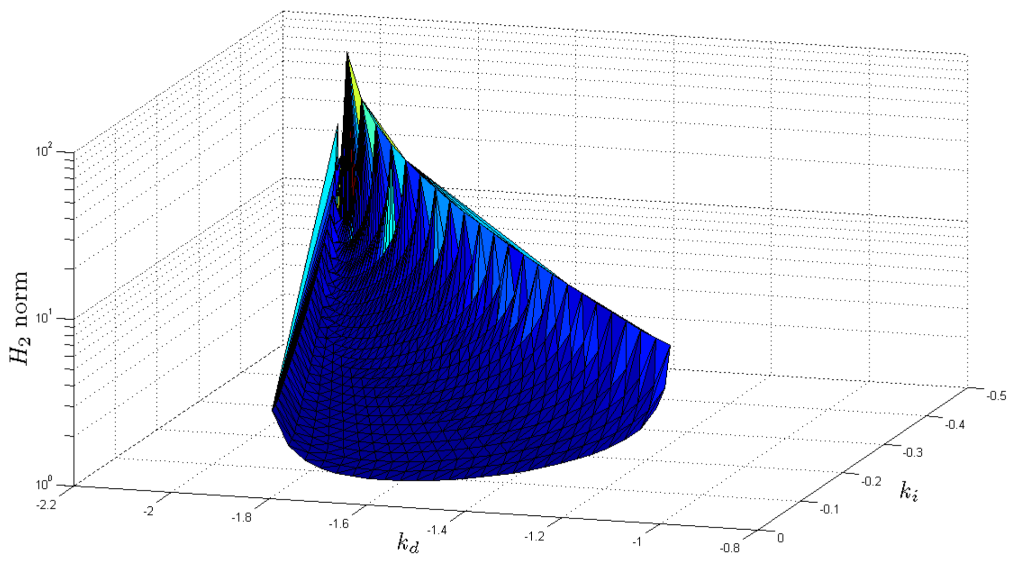

Figure 3.26: H_2 norm of $W(s)G(s)S(s)$ vs. (k_i, k_d) for $k_p = -1.648548$.

Searching over the entire (k_p, k_i, k_d) space of Figure 3.22 we calculated the minimum settling time, overshoot and undershoot to be

$$\begin{aligned}
\text{Settling Time(5\%)}: &\ 30.7400, k_p = -1.632786, k_i = -0.025468, k_d = -1.016770\\
\text{Overshoot(\%)}: &\ 268.617, k_p = -1.799857, k_i = -0.612622, k_d = -2.001945\\
\text{Undershoot(\%)}: &\ 0.252394, k_p = -1.812466, k_i = -0.025643, k_d = -1.478436
\end{aligned}$$

All of the results reported in this section were based on a brute force search over the stabilizing set. Moreover, for each controller parameter the H_∞ and H_2 norms were computed along with a 100 second simulation of the closed loop transfer function with a fixed step size of 0.01 using the euler method to obtain the time domain characteristics of settling time, overshoot and undershoot. If the resolution of the quantization of each interval range is large, this can generate a significant amount of computations. For the larger data sets existing in the PID controller space, the total computation time was on the order of hours. Increasing the resolution increased the computation time to days. This would be impractical for real time systems with fast changing plants or fast response time.

3.3 Improving Search Time

One of the disadvantages with the methods presented is the brute force search technique. Based on the resolution at which the convex set is quantized a fairly large amount of data points will be produced. For example if we quantize the K_p range into 200 points and the K_i and K_d ranges into 50 data points, we will have 500,000 data points to compute on. This will be too much data when we consider that for each data point two norm calculations and a simulation must be preformed, and we also have to search through the data to find a minimum value. In a real time system implementation, time is critical and this will not be an efficient method. To solve this problem, we present a few methods to decrease the search time of the optimal controller.

3.3.1 Coarse Grained Search to Fine Grained Search

A possible way of improving the search time would be to begin with a very coarse quantization of the parameter space and perform a brute force search. Because it is a coarse quantization there will be fewer data points to consider resulting in a quicker search. The minimum point found in the coarse search is used as the center of a square region around the point. This square region is then quantized more finely. Then a brute force search is performed on this localized region. This has the advantage of decreasing the total number of data points computed upon.

3.3.2 Gradient Descent Search

Gradient descent is a popular method for finding the minimum of a set of data quickly. Unfortunately, the data must already be present or a formula representing the gradient slope of the data should be available. In our case we have no gradient function and no data. Instead we will compute localized data and follow the localized gradient until a minimum is reached.

First, compute a quantization of the parameter space, then pick a point in the quantization set. For PID there are 27 directions a point can move to in a quantized 3D space if we include diagonal trajectories. For PI we only have 9 directions if we include the diagonals. By computing the performance metric for each of these directions around the point we can move to the next point in the direction of the largest negative change, which is the direction that is decreasing the fastest. Iteratively moving along this path will converge to a minimum.

3.3.3 Multi-core Design and Hybrid Methods

The multi-processor paradigm has been a popular way to increase the efficiency of algorithms. Because the computation of each data point is independent of every other data point, the problem is naturally amenable to a parallel implementation under the work sharing paradigm. If M processing cores are available then the total data could be divided M ways and a brute force search applied, reducing the search time by a factor[3] of M. Each of the previous methods for reducing search time could use the multi-core paradigm to improve search speed and accuracy or hybrid versions of these search methods can be conceived.

If we reconsider the coarse to fine grained search again we can develop a method of improving its accuracy without sacrificing too much time by exploiting a parallel

[3]in the ideal case. There is of course some software and hardware overhead in the implementation

algorithm. Let $k = M$ and during the coarse grained search we store the k best controllers. Next, each processing core performs the fine grained local search in parallel at each of the k regions around the k points found. Then the overall minimum can be selected from the set of k best controllers found.

In the Gradient Search method we could choose M random points and have each processing core in parallel determine a local minimum and then out of the M local minimum the best controller can be chosen.

Additionally, we can extend these search methods into hybrid methods by taking elements from one search idea and combine them with elements of another. For example, we could use a coarse grained gradient descent search, and then apply a fine grained brute force search in a square region around the minimum point found in the coarse search. Or, we could use a coarse grained brute force search and then apply a fine grained gradient descent search in the square region around the point found. The number of hybrid search methods based on the two search ideas presented here in combination with parallel algorithm paradigms is plentiful.

3.3.4 Implementation Results of Fast Search Methods for P, PI, and PID Controllers

The search methods implemented and compared in this section are a Brute Force Search(BFS), Coarse Grained to Fine Grained Search(C2FS), and Gradient Descent Search(GDS). All these search methods are performed on the same plant model and weighting function used in this chapter,

$$\begin{aligned} G(s) &= \frac{N(s)}{D(s)} = \frac{s-1}{s^2+0.8s-0.2} \\ W(s) &= \frac{s+0.1}{s+1} \ . \end{aligned}$$

These search algorithms were implemented on the TMDSEVM6678LE Evaluation Module, the same test system used for all the algorithms in this dissertation. Timing results were obtained by using the embedded time stamp counter registers TSCL and TSCH [43]. These are CPU registers that contain a free running 64-bit counter that advances at each CPU clock. By profiling the code for each algorithm the number of clock cycles spent on a particular code segment can be determined. This value can then be divided by the clock frequency of the DSP hardware to obtain the "wall clock" time spent executing the code segment.

For the constant gain stabilization results, we quantized the K_p region into 300 data points and performed a BFS for the minimum H_∞ and H_2 norms. For the GDS we also use the same 300 data points and we begin the search at the mid-point of the K_p interval. In the C2FS we use a coarse quantization of 50 data points and a fine quantization of 50 data points, producing a total search of 100 data points. The timing results are present in table 3.3 along with the minimum value calculated through each respective search. We find that the values calculated for each metric are approximately the same and the advantages in processing time is dampened by the small set of data available to compute on. Larger variations in timing data will result when the data set sizes increase under the PI and PID controllers. However, we still see improvements in the timing of the search methods over BFS while maintaining consensus in the optimal value returned.

Table 3.3: Timing of each search method for constant gain control.

	BFS	C2FS	GDS	
H_∞	1.20189	0.49767	0.73423	Time(sec)
	0.52016	0.51892	0.52273	Value
H_2	0.92607	0.29722	0.51382	Time(sec)
	1.03831	1.03830	1.03831	Value

In PI control we quantized the K_p region into 300 data points and the K_i region into 200 data points, for a total of 60,000 data points and performed a BFS for the minimum H_∞ and H_2 norms, settling time, overshoot, and undershoot. For the GDS we used the same 60,000 data points and begin the search at the best "non-fragile" controller defined by the center of mass calculation. For the C2FS we quantize the coarse search to 50 data points for both the K_p and K_i intervals and quantize the fine search to 50 data points for both the K_p and K_i intervals inside the localized region. This amounts to only 5,000 data points; much less than the BFS approach. The timing results are shown in table 3.4 along with the minimum value computed through each respective search.

Table 3.4: Timing of each search method for PI control.

	BFS	C2FS	GDS	
H_∞	968.037	137.713	12.457	Time(sec)
	0.930681	0.930437	0.949214	Value
H_2	544.456	62.252	5.845	Time(sec)
	1.18434	1.18386	1.18482	Value
Settle Time	1875.798	367.990	3.718	Time(sec)
	21.8	21.14	35.0	Value
Overshoot	1875.798	227.210	49.955	Time(sec)
	154.8806	154.8451	154.881	Value
Undershoot	1875.798	256.550	34.413	Time(sec)
	10.1529	10.0962	12.8025	Value

With these results we discover the power of these faster search algorithms. We first note that the values of each performance metric found is relatively close in value for all the metrics and nearly the same for H_∞ and H_2 norms. The outlier in the data is the settling time metric for GDS. If we refer back to figure 3.6 we will notice that the gradient of the plot seems to make a zig-zag down to the minimum point. This apparent zig-zag may be producing local minima that prevent the GDS from reaching the true minimum. Because of this, it is not advisable to use a GDS method to find minimum settling times. Excluding the settling time data, the GDS performs the fastest search and provides accurate results compared to the BFS.

In PID control we quantized the K_p region into 100 data points and the K_i and K_d regions into 50 data points each, for a total of 250,000 data points and performed a BFS for the minimum H_∞ and H_2 norms, settling time, overshoot, and undershoot.

For the GDS we used the same 250,000 data points and begin the search at the best "non-fragile" controller defined by the largest inscribed circle calculation. For the C2FS we quantize the coarse search to 10 data points for the K_p, K_i, and K_d intervals and quantize the fine search in the same manor for the localized region. This amounts to only 2,000 data points, much less then the BFS approach. The timing results are present in table 3.5 along with the minimum values calculated through each respective search.

Table 3.5: Timing of each search method for PID control.

	BFS	C2FS	GDS	
H_∞	121420	85.792	150.828	Time(sec)
	0.560194	0.547988	0.521685	Value
H_2	83354.9	32.006	89.482	Time(sec)
	1.031678	1.028096	1.05423	Value
Settle Time	49171.6	232.552	114.728	Time(sec)
	0.24	0.58	2.08	Value
Overshoot	49171.6	232.492	83.494	Time(sec)
	28.5348	31.6576	28.5348	Value
Undershoot	49171.6	150.250	150.989	Time(sec)
	10.3861	11.0232	10.3861	Value

Again, with the exception of settling time under GDS, the faster search methods still maintain accurate results while keeping search time extremely low compared to BFS. With a search space as large as 250,000 points the BFS took on the order of 10's of hours to complete, while the GDS over the same point space size reduced the search time to less than three minutes. Also, surprising results are found in C2FS with a search space of only 2,000 data points. The value of the results are a little less accurate than BFS and GDS, but the timing results surpass the other two search methods. The drastically reduced search space is what causes C2FS to beat the timing in GDS, which was not the case in PI control previously.

These faster search methods give viability to the algorithms presented in chapter 2 for use in real time auto tuning of PID controllers. These methods are also extendable to the different class of systems to be covered next.

3.4 Summary

In this chapter we applied the algorithms of chapter 2 to help solve PID optimal control problems. Fast and efficient algorithms were developed and chosen to calculate necessary results such as H_2 and H_∞ norms of the closed loop transfer function. Other criteria evaluated were settling time, overshoot, and undershoot. These algorithms were tested against numerous examples. A brute force search technique for auto-tuning of PID controllers in real time proved to be unusable and thus two new search techniques, Gradient Descent Search and Coarse to Fine Grained Search, were formulated. These faster search techniques made considerable improvements in search speed over brute force search and made the application of the algorithms of chapter 2 possible for real time systems.

In the next chapter we will expand our embedded systems algorithms to the use of non-fragile and robust controllers as formulated in [24].

4. ROBUST AND NON-FRAGILE CONTROLLER DESIGN

In this chapter we examine the TiDSP C++ implementation of the covered algorithms for the design of robust and non-fragile controllers. We are dealing with robust control because the plants are uncertain, and in our case, this uncertainty is modeled using interval plants. An interval plant is a standard polynomial plant model with real coefficients, but the coefficients are now known to vary within some closed intervals of $\mathbb{R}$. Under this scenario, the structure of the plant model, as in the degrees of the numerator and denominator polynomials, will not change but the coefficient values are allowed to perturb within some known closed intervals. Kharitonov's celebrated theorem provides necessary and sufficient conditions for the Hurwitz stability of interval polynomials, but cannot be applied directly when we are combining a controller with an interval plant. In [40], Chapellat, Bhattacharyya, and Keel, develop a generalized form of Kharitonov's theorem, which is a crucial step in applying the previous results to this new problem domain.

4.1 Robust Constant Gain Stabilization of Interval Plants

The solution to the constant gain stabilization problem involves applying the Kharitonov theorem by finding the four Kharitonov polynomials for both the numerator and the denominator of the plant. This creates eight new polynomials in total and sixteen total plant models to consider. For completeness we restate the definition of an interval polynomial and Kharitonov's theorem as presented in [24].

Consider a set $\mathcal{F}$ of all real polynomials of degree n of the form

$$P(s) = p_0 + p_1 s + p_2 s^2 + \cdots + p_n s^n$$

where the coefficients vary in independent intervals

$$p_0 \in [x_0, y_0], p_1 \in [x_1, y_1], \ldots, p_n \in [x_n, y_n], 0 \notin [x_n, y_n]$$

Such a set of polynomials is called an interval polynomial. Kharitonov's theorem states that every polynomial in the interval family $\mathcal{F}$ is Hurwitz if and only if the following four polynomials called Kharitonov polynomials are Hurwitz:

$$K^1(s) = x_0 + x_1 s + y_2 s^2 + y_3 s^3 + x_4 s^4 + x_5 s^5 + y_6 s^6 + \ldots$$
$$K^2(s) = x_0 + y_1 s + y_2 s^2 + x_3 s^3 + x_4 s^4 + y_5 s^5 + y_6 s^6 + \ldots$$
$$K^3(s) = y_0 + x_1 s + x_2 s^2 + y_3 s^3 + y_4 s^4 + x_5 s^5 + x_6 s^6 + \ldots$$
$$K^4(s) = y_0 + y_1 s + x_2 s^2 + x_3 s^3 + y_4 s^4 + y_5 s^5 + x_6 s^6 + \ldots$$

To illustrate Kharitonov's theorem, we consider the following example:

Ex. 5 — Let $P(s) = b_4 s^4 + b_3 s^3 + b_2 s^2 + b_1 s + b_0$ where $b_4 \in [1, 1], b_3 \in [3, 4], b_2 \in [4, 4], b_1 \in [5, 8], b_0 \in [6, 7]$. Determine the four Kharitonov Polynomials corresponding to the plant $P(s)$.

Answer (Ex. 5) — The four Kharitonov polynomials corresponding to this plant are:

$$K^1(s) = s^4 + 4s^3 + 4s^2 + 5s + 6$$
$$K^2(s) = s^4 + 3s^3 + 4s^2 + 8s + 6$$
$$K^3(s) = s^4 + 4s^3 + 4s^2 + 5s + 7$$
$$K^4(s) = s^4 + 3s^3 + 4s^2 + 8s + 7\ .$$

The constant gain stabilization algorithm to characterize the entire stabilizing set of k_p values for an interval plant can now be stated as follows:

1. Determine the four Kharitonov polynomials for both $N(s)$ and $D(s)$.
2. For each of the 16 combinations of $G(s) = \frac{N^i(s)}{D^j(s)}, i, j \in \{1, 2, 3, 4\}$ execute the [P] algorithm and get the respective K_p range.
3. Return the intersection of the 16 K_p ranges found.

The main computing arm of this algorithm is in the already discussed [P] algorithm. By simply constructing the 16 plants from the Kharitonov polynomials and finding the K_p intervals for each plant and then taking the intersection of all the K_p intervals, the stabilizing set is obtained.

4.1.1 Computing the Intersection of a Set of Intervals

Outside of constructing the Kharitonov polynomials, the major addition to the TiDSP C++ implementation is an algorithm to find the intersection of a set of intervals on the real line.

Computationally, an interval on our platform is represented as two floating point double precision numbers a and b with $a < b$. The left end point (LEP) is a and the right end point (REP) is b. When we consider a set of intervals we will add numerical subscripts to a and b. For example, the sixteen combinations of plants produce sixteen intervals and we denote the sets as $\{(a_i, b_i) | i \in [1, 2, \ldots, 16]\}$. The algorithm presented here requires two phases. Phase one is a search on the left end points first, followed by the right, and phase two reverses this search. The algorithm then compares the two intervals found from both phases and determines their validity. The details of this are found in Algorithm 6.

Algorithm 6 INTERVAL_INTERSECTION()

Precondition: The two passes produce $I_1 = (a_{I_1}, b_{I_1})$ and $I_2 = (a_{I_2}, b_{I_2})$. Let $c_1 \in I_1$ and $c_2 \in I_2$.

```
function INTERVAL_INTERSECTION()
    $a_{I_1} \leftarrow \max_{a_i}(a_i, b_i)\ \forall i$
    $b_{I_1} \leftarrow \min_{b_i > a_{I_1}}(a_i, b_i)\ \forall i$
    $b_{I_2} \leftarrow \min_{b_i}(a_i, b_i)\ \forall i$
    $a_{I_2} \leftarrow \max_{a_i \leq b_{I_2}}(a_i, b_i)\ \forall i$
    if $I_1 = I_2$ then
        Return $I_1$
    if $c_1 \in \{(a_i, b_i)\ \forall i\}$ and $c_2 \in \{(a_i, b_i)\ \forall i\}$ then
        Return $I_1$ and $I_2$
    if $c_1 \in \{(a_i, b_i)\ \forall i\}$ then
        Return $I_1$
    if $c_2 \in \{(a_i, b_i)\ \forall i\}$ then
        Return $I_2$
```

4.2 Robust PI Control of Interval Plants

The application of the [PI] algorithm to interval plants is straight forward. The algorithm follows along similar lines as the constant gain stabilization algorithm for interval plants with the additional need to narrow the K_p sweeping range. The algorithm is detailed below.

1. Determine the 4 Kharitonov polynomials for both $N(s)$ and $D(s)$.

2. For each of the 16 combinations of $G(s) = \frac{N^i(s)}{D^j(s)}, i, j \in \{1, 2, 3, 4\}$ execute the K_p range narrowing algorithm (step [PI].4 from the [PI] algorithm) and get the respective narrowed K_p range.

3. Form the intersection of the 16 narrowed K_p ranges as the global K_p range.

4. For each k_p in the globally narrowed K_p range do the following

a) For each of the 16 combinations of $G(s)$ run the [PI] algorithm to find the K_i interval.

b) Take the intersection of the 16 K_i intervals to produce the global K_i interval for that specific k_p.

This algorithm does not present any new challenges for the existing embedded systems implementation.

4.3 Robust PID Control of Interval Plants

For robust stabilization of interval plants under PID controllers the Generalized Kharitonov Theorem [40] is invoked. This introduces the concept of the four Kharitonov segments [24] of $N(s)$. Let $N^i(s), i = 1, 2, 3, 4$ and $D^j(s), j = 1, 2, 3, 4$ be the Kharitonov polynomials corresponding to $N(s)$ and $D(s)$, respectively. Now, let $NS^i(s), i = 1, 2, 3, 4$ be the four Kharitonov segments of $N(s)$ where

$$\begin{aligned}
NS^1(s,\lambda) &= (1-\lambda)N^1(s) + \lambda N^2(s) \\
NS^2(s,\lambda) &= (1-\lambda)N^1(s) + \lambda N^3(s) \\
NS^3(s,\lambda) &= (1-\lambda)N^2(s) + \lambda N^4(s) \\
NS^4(s,\lambda) &= (1-\lambda)N^3(s) + \lambda N^4(s)
\end{aligned}$$

and $\lambda \in [0, 1]$. Let $G_S(s)$ denote the family of 16 segment plants:

$$G_S(s) = \{G_{ij}(s,\lambda) | G_{ij}(s,\lambda) = \frac{NS^i(s,\lambda)}{D^j(s)}, i = 1,2,3,4, j = 1,2,3,4, \lambda \in [0,1]\} \ .$$

Thus, to characterize all stabilizing PID controllers for the entire family of segment plants $G_S(s)$, the entire family of closed loop characteristic polynomials must be Hurwitz. Using the already established [PID] algorithm and the Generalized Kharitonov

Theorem, the robust stabilization algorithm achieves Hurwitz stability of the entire family of closed loop characteristic polynomials.

1. Determine the 4 Kharitonov polynomials for both $N(s)$ and $D(s)$.

2. For each of the 16 combinations of $G(s) = \frac{N^i(s)}{D^j(s)}, i, j \in \{1, 2, 3, 4\}$ execute the K_p range narrowing algorithm and get the respective narrowed K_p range.

3. Take the intersection of the 16 narrowed K_p ranges as the global K_p range.

4. For each k_p in the globally narrowed K_p range do the following

   a) For each of the 16 segment plants $G_S(s)$

      - Sweep over $\lambda \in [0, 1]$ and execute the [PID] algorithm and store the set of linear inequalities that define the (k_i, k_d) convex set for each λ.

      - Find the intersection of the set of linear inequalities to define the convex set over all λ values.

   b) Find the Feasible Region of all the linear inequalities generated by the 16 segment plants.

   c) Quantize the (k_i, k_d) region.

   d) Solve the largest inscribed circle.

5. Find the largest radius circle over all the k_p values and return the optimal "non-fragile" controller.

In step 4a we look to find the solution of the feasible region for a given segment plant over all values of λ, but we do not want to enumerate the vertices. Instead, we want to find the minimal set of linear inequalities that will define the feasible region over the λ sweep. This allows us to then combine all the linear inequalities after processing all the segment plants to then solve for the feasible region via vertex enumeration for each k_p value. Thus, we make a small modification to the `FEASIBLE_REGION()` algorithm previously discussed. The modification saves the linear inequality associated with each vertex found during the search for later use.

The concept of "non-fragile" controllers is discussed in [24]. A controller that destabilizes the system due to small perturbations in the controller parameters is said to be "fragile." The practical importance of this idea stems from the fact that any implementation of the controller parameters will never be ideal. Thus, to design for fragility we can choose the controller that is furthest away from the stability boundaries. Because we have characterized the entire region of PID controllers that stabilize a system, this is now an easy task. The exact solution to this comes from our previously described solution of finding the largest inscribed circle. By reusing this solution method we are able to find the optimal "non-fragile" PID controller for any given interval plant.

4.4 Examples and Results

In this section we verify all the examples in [24] involving interval plants on our embedded system implementation. In all cases we consider the following plant model

$$G(s) = \frac{N(s)}{D(s)}$$

$$N(s) = a_0 + a_1 s + a_2 s^2 + a_3 s^3 + \ldots$$

$$D(s) = b_0 + b_1 s + b_2 s^2 + b_3 s^3 + \ldots$$

where the particular a_i and b_i parameters are defined for each example.

4.4.1 Robust Stabilization Using Constant Gain

For this example we let

$$a_2 \in [1,1], a_1 \in [1,2], a_0 \in [1,2]$$
$$b_4 \in [1,1], b_3 \in [3,4], b_2 \in [4,4], b_1 \in [5,8], b_0 \in [6,7] \ .$$

Running the [P] algorithm for Robust Stabilization on our embedded system produces the following Kharitonov polynomials:

$$N^1(s) = 1.0s^2 + 1.0s + 1.0$$
$$N^2(s) = 1.0s^2 + 2.0s + 1.0$$
$$N^3(s) = 1.0s^2 + 1.0s + 2.0$$
$$N^4(s) = 1.0s^2 + 2.0s + 2.0$$
$$D^1(s) = 1.0s^4 + 4.0s^3 + 4.0s^2 + 5.0s + 6.0$$
$$D^2(s) = 1.0s^4 + 3.0s^3 + 4.0s^2 + 8.0s + 6.0$$
$$D^3(s) = 1.0s^4 + 4.0s^3 + 4.0s^2 + 5.0s + 7.0$$
$$D^4(s) = 1.0s^4 + 3.0s^3 + 4.0s^2 + 8.0s + 7.0 \ .$$

The set of intervals generated for each of the 16 vertex plants is:

$$
\begin{aligned}
K_{11} &= (2.388508, \infty) \\
K_{12} &= (1.558422, \infty) \\
K_{13} &= (3.000000, \infty) \\
K_{14} &= (2.052343, \infty) \\
K_{21} &= (1.774917, \infty) \\
K_{22} &= (2.000000, \infty) \\
K_{23} &= (2.272002, \infty) \\
K_{24} &= (2.558422, \infty) \\
K_{31} &= (-3.00000, -2.829708) \cup (4.829708, \infty) \\
K_{32} &= (2.854102, \infty) \\
K_{33} &= (-3.500000, -3.472136) \cup (5.472136, \infty) \\
K_{34} &= (3.468627, \infty) \\
K_{41} &= (3.201562, \infty) \\
K_{42} &= (-3.00000, -2.854102) \cup (3.854102, \infty) \\
K_{43} &= (3.774917, \infty) \\
K_{44} &= (-3.50000, -3.468627) \cup (4.468627, \infty) \ .
\end{aligned}
$$

Performing the intersection of all these intervals gives the final stabilizing set of constant gain controllers for this interval plant as

$$k_p \in (5.472136, \infty) \ .$$

It is impossible to verify the stability of the system over all possible k_p values, but as a representative example to the systems stability and correctness of the algorithm we consider an arbitrary plant within the coefficient intervals specified and choose an arbitrary k_p value close to the left end point of the K_p interval, and simulate the system using the MATLAB SIMULINK model of Figure 2.1.

The arbitrary plant that was generated for this example is

$$N(s) = 1.0000s^2 + 1.3532s + 1.8212$$

$$D(s) = 1.0000s^4 + 3.0430s^3 + 4.0000s^2 + 6.9473s + 6.7317$$

and the arbitrary k_p value is $k_p = 6.1497$. The output response is presented in Figure 4.1.

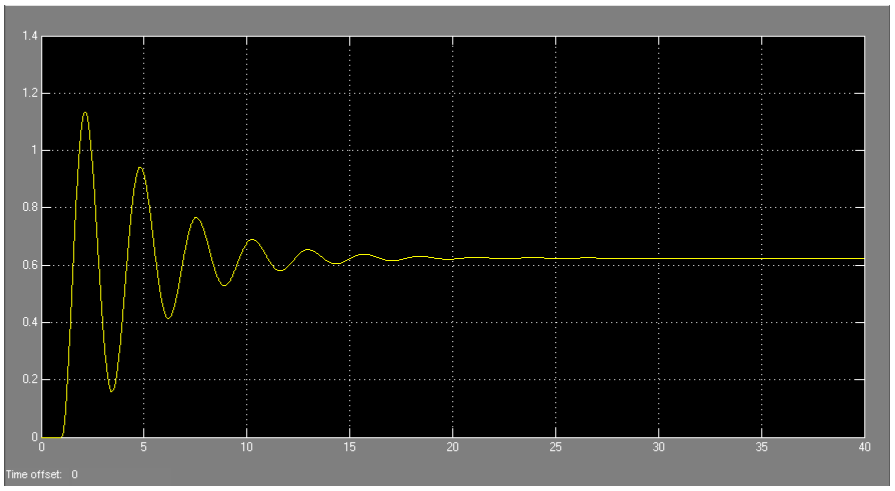

Figure 4.1: System response with $k_p = 6.1497$.

4.4.2 Robust Stabilization Using PI Control

For this example we let

$$a_4 \in [1,1], a_3 \in [2,3], a_2 \in [39,41], a_1 \in [48,50], a_0 \in [-6,-3]$$

$$b_5 \in [1,1], b_4 \in [2,3], b_3 \in [31,32], b_2 \in [35,38], b_1 \in [49,51], b_0 \in [97,101] \ .$$

Running the [PI] algorithm for Robust Stabilization on our embedded systems produces the following Kharitonov polynomials:

$$N^1(s) = 1.0s^4 + 3.0s^2 + 41.0s^2 + 48.0s - 6.0$$

$$N^2(s) = 1.0s^4 + 2.0s^2 + 41.0s^2 + 50.0s - 6.0$$

$$N^3(s) = 1.0s^4 + 3.0s^2 + 39.0s^2 + 48.0s - 3.0$$

$$N^4(s) = 1.0s^4 + 2.0s^2 + 39.0s^2 + 50.0s - 3.0$$

$$D^1(s) = 1.0s^5 + 2.0s^4 + 32.0s^3 + 38.0s^2 + 49.0s + 97.0$$

$$D^2(s) = 1.0s^5 + 2.0s^4 + 31.0s^3 + 38.0s^2 + 51.0s + 97.0$$

$$D^3(s) = 1.0s^5 + 3.0s^4 + 32.0s^3 + 35.0s^2 + 49.0s + 101.0$$

$$D^4(s) = 1.0s^5 + 3.0s^4 + 31.0s^3 + 35.0s^2 + 51.0s + 101.0 \ .$$

Performing the K_p narrowing routine on this examples produces the finite K_p sweeping interval to be:

$$k_p \in (0.002203, 16.166667) \ .$$

The set of intervals generated for each of the 16 vertex plants for the specific value of $k_p = 2.0$ are

$$
\begin{aligned}
K_{11} &= (-1.499299, -0.000000) \\
K_{12} &= (-1.504891, -0.000000) \\
K_{13} &= (-1.567277, -0.000000) \\
K_{14} &= (-1.573404, -0.000000) \\
K_{21} &= (-1.460975, -0.000000) \\
K_{22} &= (-1.465634, -0.000000) \\
K_{23} &= (-1.528074, -0.000000) \\
K_{24} &= (-1.533151, -0.000000) \\
K_{31} &= (-1.735555, -0.000000) \\
K_{32} &= (-1.739353, -0.000000) \\
K_{33} &= (-1.809986, -0.000000) \\
K_{34} &= (-1.814157, -0.000000) \\
K_{41} &= (-1.680651, -0.000000) \\
K_{42} &= (-1.683727, -0.000000) \\
K_{43} &= (-1.753380, -0.000000) \\
K_{44} &= (-1.756733, -0.000000) \ .
\end{aligned}
$$

Sweeping over all k_p in the valid range results in the stabilizing region depicted in Figure 4.2. We then compute the center of mass for each of the disjoint 2D regions to find the optimal "non-fragile" controllers. The two candidate controllers were found to be

$$k_p^{(1)} = 1.507087 \qquad k_p^{(2)} = 13.391186$$
$$k_i^{(1)} = -0.72247 \qquad k_i^{(2)} = 0.153593 \ .$$

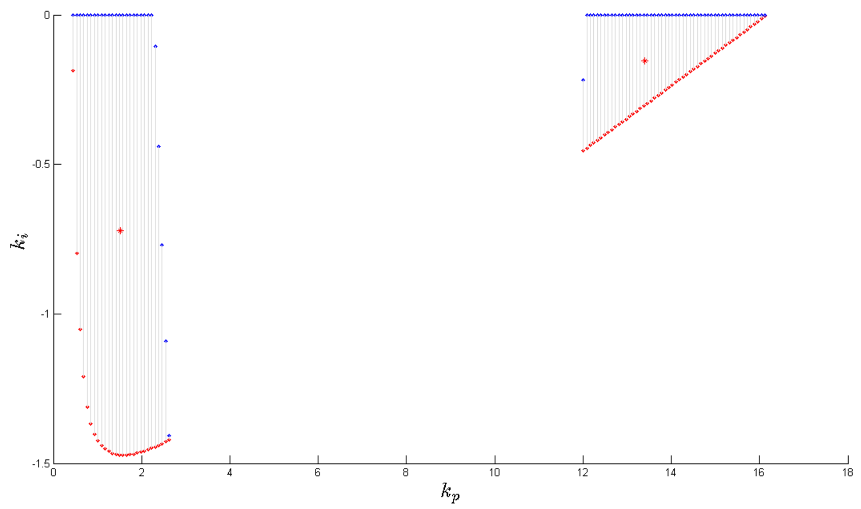

Figure 4.2: The stabilizing set of (k_p, k_i) values with “non-fragile” controllers indicated by the star.

Even though verifying the entire stabilizing set of (k_p, k_i) pairs would be impossible, we will consider a representative example to this solutions validity by considering an arbitrary plant within the coefficient intervals specified and choose the (k_p, k_i) pair given by the center of mass calculation on the region and simulate the system using the MATLAB SIMULINK model of Figure 2.4.

The arbitrary plant that was created for this example is

$$N(s) = 1.0000s^4 + 2.5470s^3 + 39.5926s^2 + 49.4894s - 5.4331$$
$$D(s) = 1.0000s^5 + 2.1835s^4 + 31.3685s^3 + 36.8769s^2 + 50.5605s + 97.3245$$

and the (k_p, k_i) pair is $k_p = 1.5071$ and $k_i = -0.7225$. The output response is presented in Figure 4.3.

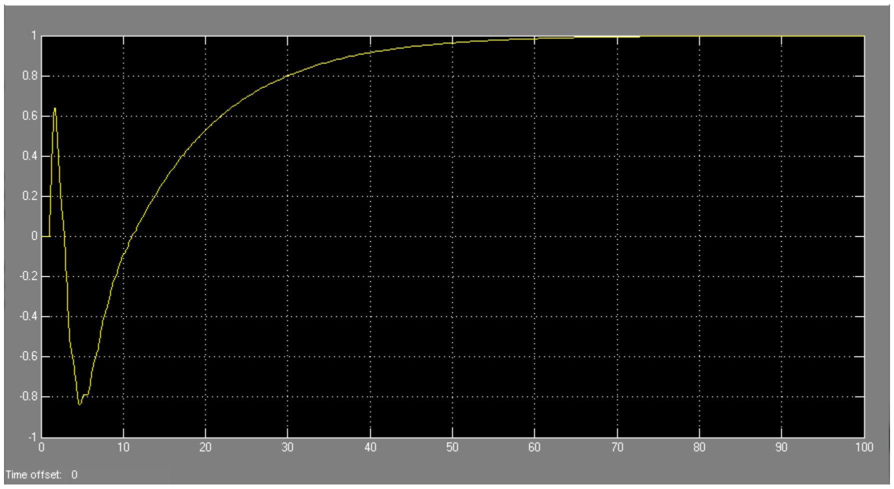

Figure 4.3: System response with $k_p = 1.5071$ and $k_i = -0.7225$.

Remark: In the case of disjoint sets, the center of mass calculation can fail to give a result that is contained in the stability region. To alleviate this, we calculate the center of mass for each disjoint set. Then a simple metric can be chosen to determine which one is the better choice.

4.4.3 Robust Stabilization Using PID Control

For this example we let

$$a_2 \in [1,1], a_1 \in [-5,-4], a_0 \in [2,4]$$

$$b_5 \in [1,1], b_4 \in [3,4], b_3 \in [5,5], b_2 \in [7,9], b_1 \in [8,9], b_0 \in [-2,-1] \ .$$

Running the [PID] algorithm for Robust Stabilization on our embedded systems produces the following Kharitonov polynomials:

$$N^1(s) = 1.0s^2 - 5.0s + 2.0$$

$$N^2(s) = 1.0s^2 - 4.0s + 2.0$$

$$N^3(s) = 1.0s^2 - 5.0s + 4.0$$

$$N^4(s) = 1.0s^2 - 4.0s + 4.0$$

$$D^1(s) = 1.0s^5 + 3.0s^4 + 5.0s^3 + 9.0s^2 + 8.0s - 2.0$$

$$D^2(s) = 1.0s^5 + 3.0s^4 + 5.0s^3 + 9.0s^2 + 9.0s - 2.0$$

$$D^3(s) = 1.0s^5 + 4.0s^4 + 5.0s^3 + 7.0s^2 + 8.0s - 1.0$$

$$D^4(s) = 1.0s^5 + 4.0s^4 + 5.0s^3 + 7.0s^2 + 9.0s - 1.0 \ .$$

Applying the K_p narrowing routine to this examples produces the finite K_p sweeping interval:

$$k_p \in (1.000000, 1.086872) \ .$$

For a fixed k_p value we can examine some of the preliminary results of the algorithm. Let $k_p = 1.05$ and observe in Figure 4.4 the stabilizing regions for $G_{11}(s, \lambda)$ for different values of $\lambda \in [0, 1]$. We chose the quantized set of

$$\lambda \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\} \ .$$

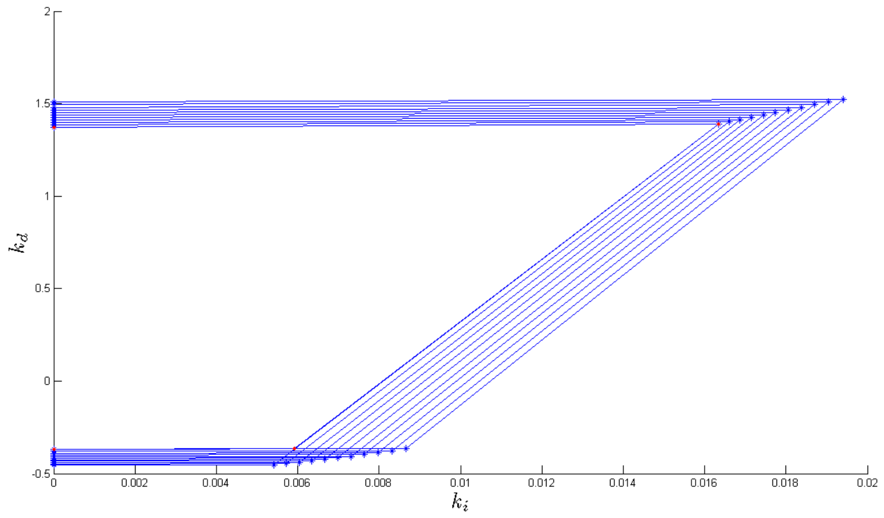

Figure 4.4: The stabilizing set of (k_i, k_d) values for $G_{11}(s, \lambda)$, $\lambda \in [0, 1]$ with $k_p = 1.05$.

The red dots indicate the vertex points of the intersection of all the regions. By continuing the execution of the algorithm over all 16 segment plants for the fixed k_p value we obtain the set of regions in Figure 4.5, with the red circles indicating the vertex points of the intersection of all the regions.

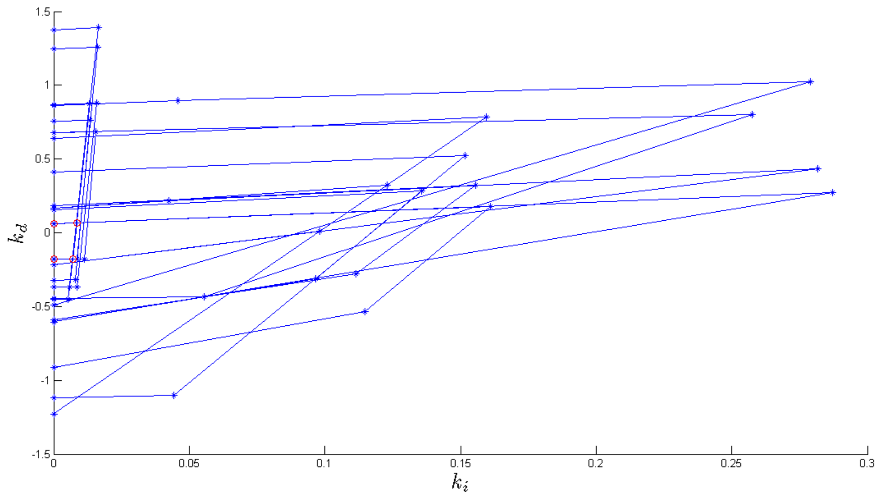

Figure 4.5: The stabilizing set of (k_i, k_d) values for $G_{ij}(s, \lambda)$, $\lambda \in [0, 1]$, $i = 1, 2, 3, 4$ and $j = 1, 2, 3, 4$ with $k_p = 1.05$.

Executing the algorithm over all k_p in the narrowed K_p interval we obtain all stabilizing regions for each k_p value. These regions are depicted in Figure 4.6.

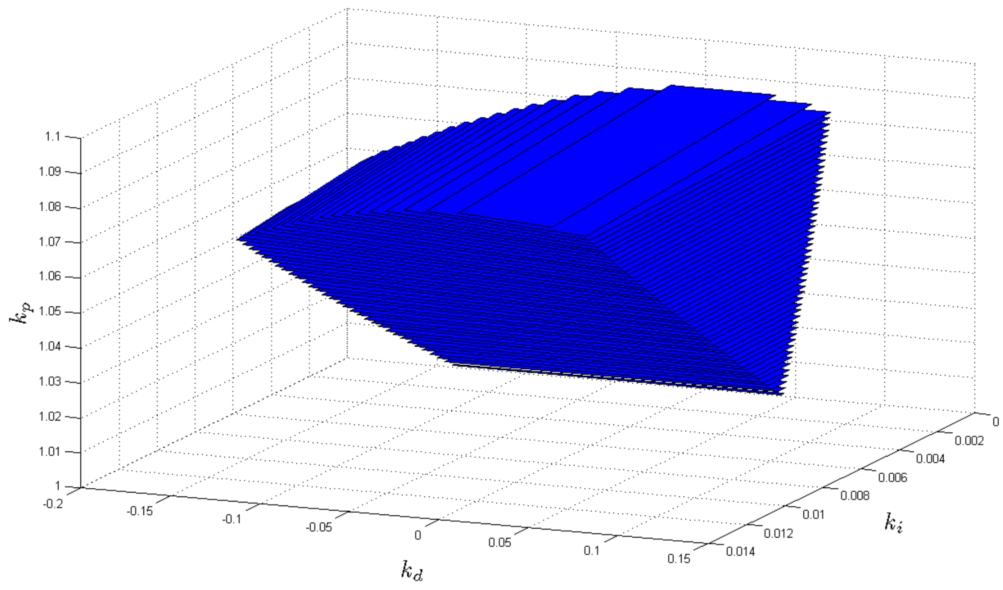

Figure 4.6: Stabilizing set of (k_p, k_i, k_d) values.

Next we perform the largest inscribed circle calculation for each k_p and search for the largest radius circle. This optimal "non-fragile" controller was found at $k_p = 1.086$, $k_i = 0.006839$, $k_d = 0.037335$ and has a corresponding circle with radius of $r = 0.006839$. A visual depiction of this region can be seen in Figure 4.7.

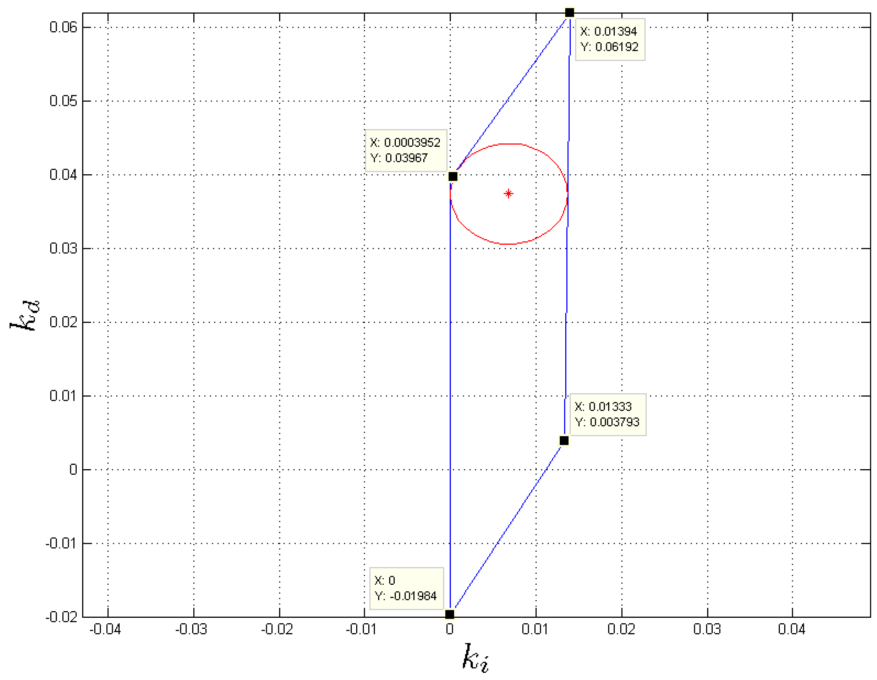

Figure 4.7: The (k_i, k_d) stabilizing region for $k_p = 1.086$ and the inscribed circle for "non-fragile" controller selection.

4.5 Summary

In this chapter we successfully extended our embedded systems implementation of the P, PI, and PID control algorithms to include robust and non-fragile controller design. This required a computational implementation of the generalized form of Kharitonov's theorem and a reuse of our previously developed algorithms from chapter 2. An additional sub algorithm to determine the intersection of a set of intervals was also developed. These new algorithms were tested against a number of examples and the results were provided. In the next chapter we will continue to expand our embedded systems algorithms to the use of controller design for first order systems with time delay.

5. STABILIZATION OF FIRST ORDER SYSTEMS WITH TIME DELAY

In most applications the systems being modeled exhibit some form of a time delay. This is a natural occurrence as most processes require time to occur. In some systems this time can be ignored, but in others we must include this time delay in the plant model for accurate control. For a first order system, mathematically, the plant model is of the form

$$G(s) = \frac{k}{1+Ts}\exp^{-Ls}$$

where k models the steady-state gain, L models the time delay, and T models the time constant of the plant. The exponential term creates closed loop characteristic equations which involve terms that are called quasipolynomials [24]. The algorithms covered thus far will not apply to these types of polynomials, but only to plants described by real rational transfer functions. Thus a suitable extension to the Hermite-Biehler Theorem is presented in [24], based on the results in [40, 44, 45]. This extension allows us to properly approach the time delay plant models and quasipolynomials to apply computational approaches for the characterization of stabilizing constant gain controllers, pure integral controllers, and proportional-integral controllers.

5.1 Constant Gain Stabilization of First Order Systems with Time Delay

There are two main results from [24] presented here: constant gain stabilization for open loop stable plants and constant gain stabilization for open loop unstable plants. These results define the path to implementation in the TiDSP C++ environment.

5.1.1 Open-loop Stable Plants

For open-loop stable plants we require that $T > 0$. We further assume that $k > 0$ and $L > 0$. Under these conditions [24] defines the set of all stabilizing gains k_c for a given first order plant with time delay as

$$-\frac{1}{k} < k_c < \frac{T}{kL}\sqrt{z_1^2 + \frac{L^2}{T^2}}$$

where z_1 is the solution of the equation

$$\tan(z) = -\frac{T}{L}z, \quad z \in (\frac{\pi}{2}, \pi) \ .$$

From this, we can see that a computational implementation of this requires only an equation solver; specifically for the equation $\tan(z) = -\frac{T}{L}z, z \in (\frac{\pi}{2}, \pi)$. This will be covered in more detail later, but first we will consider the results for the open loop unstable plants in [24] because they are closely related. This will be useful in the implementation of the equation solver.

5.1.2 Open-loop Unstable Plants

For open-loop unstable plants we require that $T < 0$ and we further assume that $k > 0$ and $L > 0$. Furthermore, a necessary condition for a gain k_c to simultaneously stabilize the delay-free plant and the plant with delay is that $|\frac{T}{L}| > 1$. Under these conditions [24] defines the set of all stabilizing gains k_c for a given first order plant with time delay as

$$\frac{T}{kL}\sqrt{z_1^2 + \frac{L^2}{T^2}} < k_c < -\frac{1}{k}$$

where z_1 is the solution of the equation

$$\tan(z) = -\frac{T}{L}z, \quad z \in (0, \frac{\pi}{2}) \ .$$

Again, we see the same equation is present as in the case of open-loop stable plants, however, the interval is shifted by $\frac{\pi}{2}$. Later, we will show how these two results can be used to create a single computational implementation that handles both the cases.

5.1.3 Computationally Finding z_1

If we observe the plots of $tan(z)$ and $-\frac{T}{L}z$ we find that in the case of open loop stable plants the line $-\frac{T}{L}z$ will always intersect the curve $tan(z)$ in the interval $(\frac{\pi}{2}, \pi$ because $T > 0$ and $L > 0$. Thus a solution to the intersection of the line and tangent curve will always exists. In the case of open loop unstable plants we have the conditions that $T < 0$, $L > 0$, and $|\frac{T}{L}| > 1$. These conditions also ensure that a solution to the intersection will always exist. This is more clearly seen given that $\frac{d}{dz}tan(z)|_0 = \cos^{-2}(z)|_0 = 1$ and $tan(z)$ is monotonically increasing to infinity in the interval of $(0, \frac{\pi}{2})$, thus a slope greater than one will always intersect such a curve.

A solution is now guaranteed under the conditions provided to us, and we can formulate a simple algorithm to find the intersection of the line and the curve. Further investigation of the plots shows that at the left side of the intersection point, the $-\frac{T}{L}z$ line is always above the $tan(z)$ curve, then this relationship reverses on the right side of the intersection point. By evaluating points from left to right we can look for a sign change in the difference of the two functions. The point at which the sign change occurs provides us with a narrow interval to search for the intersection. We can do this search repeatedly until a desired accuracy is achieved. This procedure is detailed in Algorithm 7.

Algorithm 7 FIND_Z1

```
function FIND_Z1($T, L, lp, rp, err$)
    if $T < 0$ and $|\frac{T}{L}| \leq 1$ then
        no solution
    $x \leftarrow$ Quantization of the interval $(lp, rp)$
    $y1 \leftarrow \tan(x_i)$
    $y2 \leftarrow -\frac{T}{L}x_i$
    if $y2_i < y1_i$ then
        $lp \leftarrow x_{i-1}$
        $rp \leftarrow x_i$
    Repeat until $|y2_i - y1_i| < err$ return $rp$
```

Because of the nature of the problem and the conditions imposed, this algorithm works for both open loop stable and unstable plants. To finalize, we present the algorithm for constant gain stabilization in the form of the C++ language in Listing 5.1, since it is straightforward enough without pseudo code. This algorithm is presented in Listing 5.1. We will refer to this algorithm as the [P] algorithm for time delay plants.

Listing 5.1: Algorithm for Time Delay Plants

```
std::vector<double> p_char_1otd(double k, double L, double T) {
            double PI = 3.14159265358979323846264 33;
            double z1;
            std::vector<double> r(2);

      if (T < 0) {
            z1 = FIND_Z1(T, L, 0, PI/2, 1e-6);
            r[0] = T/(k*L) * sqrt( pow(z1,2.0) + pow(L,2.0)/pow(T,2.0) );
                        r[1] = -1.0/k;
            }
      else {
            z1 = FIND_Z1(T, L, PI/2, PI, 1e-6);
            r[0] = -1/k;
                        r[1] = T/(k*L) * sqrt( pow(z1,2.0) + pow(L,2.0)/pow(T,2.0) );
            }
            return r;
}
```

5.2 Pure Integral Control of First Order Time Delay Plants

For pure integral control [24] provides a closed form solution similar to that for constant gain stabilization. This applies to open loop stable[1] plants where $T > 0$, $k > 0$, and $L > 0$. As specified in [24] the set of all stabilizing gains k_i for a given open loop stable plant with transfer function $G(s)$ is given by

$$0 < k_i < \frac{T}{kL^2} z_1 \sqrt{z_1^2 + \frac{L^2}{T^2}}$$

where z_1 is the solution to the equation

$$\frac{1}{\tan(z)} = \frac{T}{L} z, z \in (0, \frac{\pi}{2})$$

This result is closely related to the results for constant gain stabilization. One

[1]It is shown in [24] that an open loop unstable plant cannot be stabilized by pure integral control

difference is in the equation that we solve for z_1. For I-Control we solve $\frac{1}{\tan(z)} = \frac{T}{L}z$ versus $\tan(z) = -\frac{T}{L}z$ for P-Control. But with careful consideration we can reuse the same `FIND_Z1` algorithm from before. We note that $\frac{1}{tan(z)}$ and $tan(z-\frac{\pi}{2})$ are mirrored over the z-axis as is $\frac{T}{L}z$ and $-\frac{T}{L}z$. The intersection of these two function pairs are equivalent on the z-axis. Thus, finding z for $\frac{1}{tan(z)} = \frac{T}{L}z$ in $(0, \frac{\pi}{2})$ is equivalent to finding z for $tan(z - \frac{\pi}{2}) = -\frac{T}{L}z$ in $(0, \frac{\pi}{2})$. We simply need to modify the `FIND_Z1` algorithm to incorporate the option of a shift in the $\tan(z)$ argument in step 5 of the `FIND_Z1` algorithm.

5.3 PI Control of First Order Time Delay Plants

So far we have shown how to formulate the results of [24] for constant gain stabilization and pure integral control of time delay plants on a computational algorithm ready for implementation into an embedded processor. In this section we consider the results of PI control of first order time delay plants and look at the computational implementation of this algorithm. As with any computational implementation we choose to only consider the crucial parts of the theory. We can restate the algorithm presented in [24] as follows:

1. Narrow the K_p range by running the [P] algorithm for Time Delay Plants.
2. For each k_p in the narrowed range do the following:
   a) Begin loop on j.
   b) $z_j = \text{solve}(kk_p + cos(z) - \frac{T}{L}z\sin(z) = 0)$.
   c) Store $[a_j] = \frac{z_j}{kL}[\sin(z_j) + \frac{T}{L}z_j\cos(z_j)]$.
   d) If $cos(z_j) > 0$ then break the loop on j.
   e) $j = j + 2$.

3. $K_i^{k_p} = [0, \min_{q=1,3,5\ldots,j} a_q]$.

We see that in step 1 we are able to reuse our previous results to narrow down the search space of k_p values. Then for each k_p value in the narrowed space, our main computational task is to find z_j, the roots of the equation $kk_p + cos(z) - \frac{T}{L} z \sin(z) = 0$. We will be able to reuse the same algorithm concepts applied to the `FIND_Z1` algorithm. However, solving $kk_p + cos(z) - \frac{T}{L} z \sin(z) = 0$ will require finding many intersections with the z-axis. Additionally we are only concerned with every other intersection point as we traverse the z-axis from 0 to ∞. To accomplish this we create a new function to find these z_j values called `FIND_ZJ`, detailed in Algorithm 8.

Because we want to find multiple roots but do not know *a priori* how many, we design the function to return the first z_j value and record its progress along the z-axis search. This is performed in two parts. The first part finds the first sign change along the function; this indicates the first z_j value. The second part then advances along the z-axis until the next sign change occurs. It stores the point after that sign change for reuse by a future call. The theory dictates that we only consider every other root of the equation. Thus, the future call to `FIND_ZJ` will begin its search for the z_{j+2} root because the previous call to `FIND_ZJ` skipped over the z_{j+1} root.

Remark: Extension of the first-order-plant-with-time-delay results to PID controllers was accomplished after the publication of [24]. The interested reader may refer to [46] for the detailed treatment.

Algorithm 8 FIND_ZJ

Precondition: rp is passed by reference

```
function FIND_ZJ($kp, k, T, L, rp, res, ac$)
    $lp \leftarrow rp + 2\pi$
    $x \leftarrow$ Quantization of the interval $(lp, rp)$ at resolution of $res$
    $z \leftarrow kk_p + \cos(x_i) - \frac{T}{L} x_i \sin(x_i)$
    for all $z_i$ do
        if $|sgn(z_0) - sgn(z_i)| > 0$ then
            $lp \leftarrow x_{i-1}$
            $rp \leftarrow x_i$
    Repeat $ac$ times
    $z_j \leftarrow lp + \frac{rp-lp}{2}$    ▷ Find the next intersection and set $rp$ to the point after it
for use in future calls
    $lp \leftarrow rp$
    $rp \leftarrow lp + 2\pi$
    $x \leftarrow$ Quantization of the interval $(lp, rp)$ at resolution of $res$
    $z \leftarrow kk_p + \cos(x_i) - \frac{T}{L} x_i \sin(x_i)$
    for all $z_i$ do
        if $|sgn(z_0) - sgn(z_i)| > 0$ then
            $rp \leftarrow x_i$ break for loop
    return $z_j$
```

5.4 Examples and Results

In this section we verify all the examples in [24] for the case of First Order Systems with Time Delay on our embedded system implementation. In all cases we consider the following plant model

$$G(s) = \frac{k}{1+Ts}e^{-Ls}$$

where the particular numerical values for the terms k, T, and L are specified in each example. We will be using the following controllers depending on the example.

$$\begin{aligned} \text{P Control: } C(s) &= k_p \\ \text{I Control: } C(s) &= \frac{k_i}{s} \\ \text{PI Control: } C(s) &= k_p + \frac{k_i}{s} \end{aligned}$$

5.4.1 P-Control, Open-loop Stable Plant

Let $k = 1$, $L = 2$ sec, and $T = 1$ sec. Running the constant gain stabilization algorithm on the embedded system produces a stabilizing range of controllers for

$$k_p \in (-1.000000, 1.519803) \ .$$

It is impossible to verify the stability of the system over all possible k_p values, but as a representative example to the systems stability and correctness of the algorithm we consider $k_p = 0.75$ in the stabilizing range and use the MATLAB SIMULINK model of Figure 5.1 to verify the stability of the system as shown in Figure 5.2.

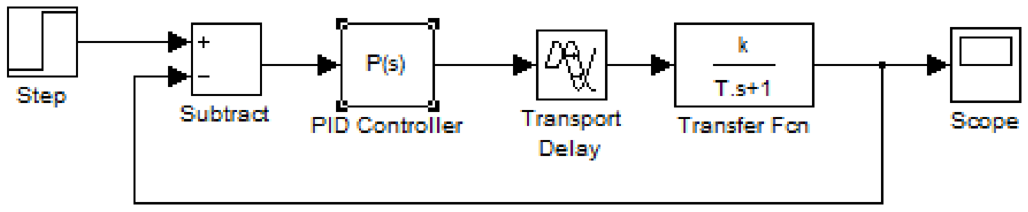

Figure 5.1: Simulation model for P-control of first order systems with time delay.

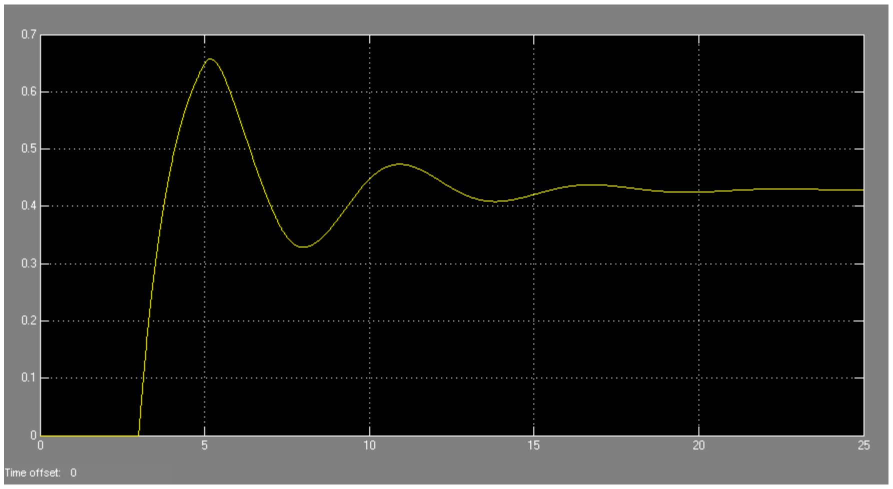

Figure 5.2: System response with $k_p = 0.75$.

5.4.2 P-Control, Open-loop Unstable Plant

Let $k = 1$, $L = 0.5$ sec, and $T = -2$ sec. Running the constant gain stabilization algorithm on the embedded system produces a stabilizing range of controllers for

$$k_p \in (-5.662004, -1.000000) \ .$$

Even though verifying the entire stabilizing set of k_p values would be impossible, we will consider a representative example to this solutions validity by considering $k_p = -3.25$ in the stable range and use the MATLAB SIMULINK model of Figure 5.1 to verify the stability of the system as presented in Figure 5.3.

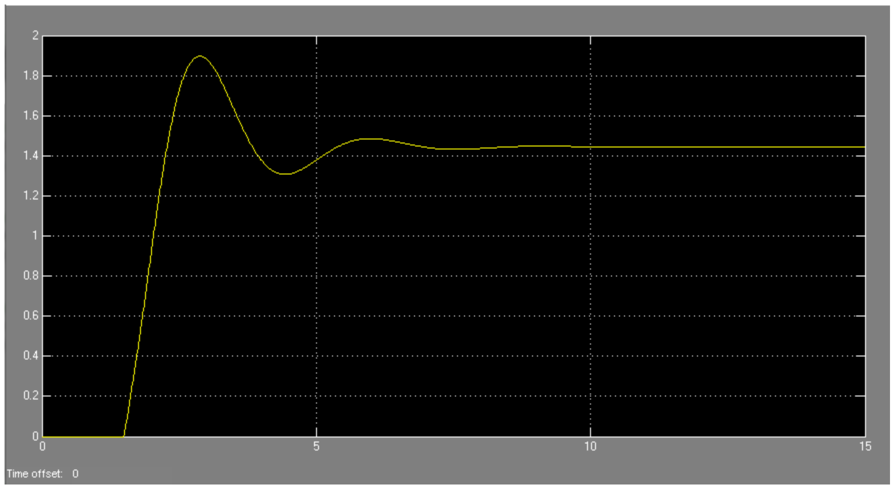

Figure 5.3: System response with $k_p = -3.25$.

5.4.3 I-Control, Open-loop Stable Plant

Let $k = 1$, $L = 1$ sec, and $T = 2$ sec. Running the constant gain stabilization algorithm on the embedded system produces a stabilizing range of controllers for

$$k_p \in (0.000000, 1.074835) \ .$$

It is not feasible to verify the stability of the system over all possible k_p values, but to provide evidence to the solutions validity we consider $k_p = 0.35$ in the stable range and use the MATLAB SIMULINK model of Figure 5.4 to verify the stability of the system as shown in Figure 5.5.

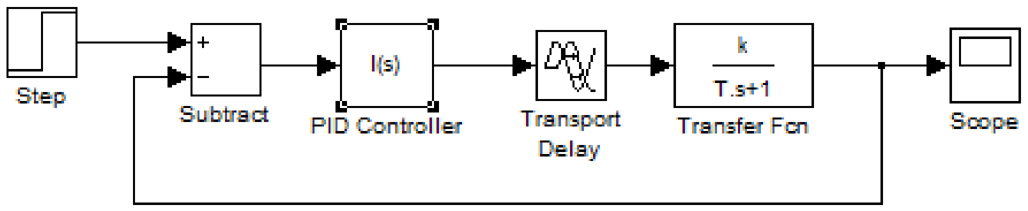

Figure 5.4: Simulation model for I-control of first order systems with time delay.

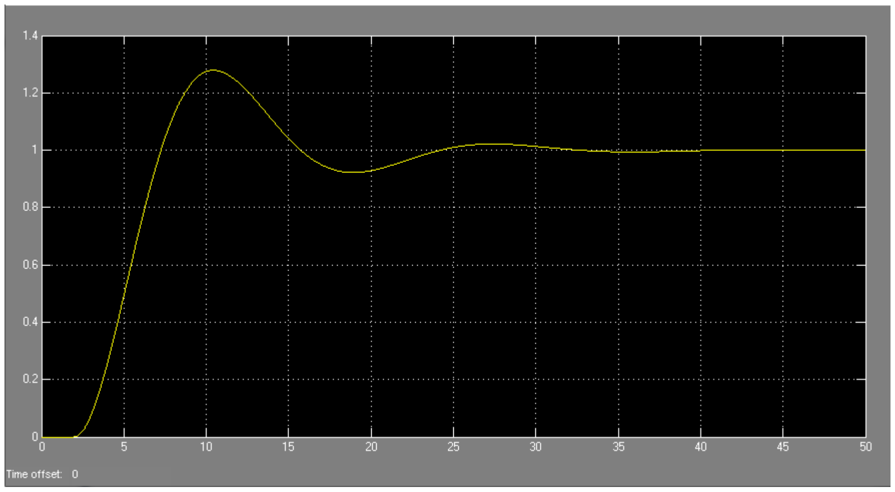

Figure 5.5: System response with $k_p = 0.35$.

5.4.4 PI-Control, Open-loop Stable Plant

Let $k = 1$, $L = 1$ sec, and $T = 4$ sec. Running the [PI] algorithm for first order systems with time delay on the embedded system produces a narrowed K_p range of

$$k_p \in (-1.000000, 6.934511) \ .$$

Sweeping over this k_p range we achieve the stability region of Figure 5.6.

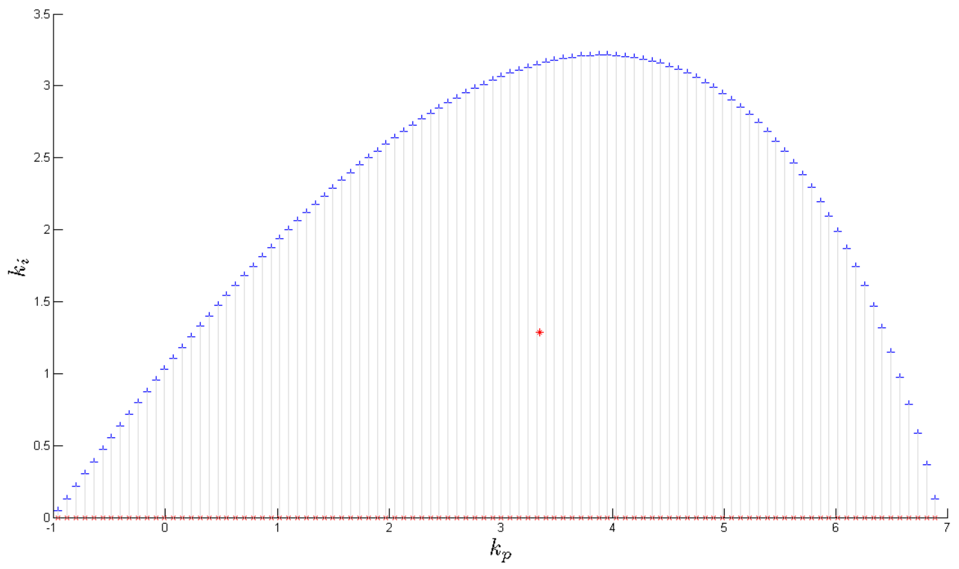

Figure 5.6: The stabilizing set of (k_p, k_i) values with "non-fragile" controller indicated by the star.

Calculating the center of mass of the 2D shape produces the "non-fragile" PI controller with

$$k_p = 3.347755$$

$$k_i = 1.287736\ .$$

The stability of the system over all possible (k_p, k_i) pairs is impossible to verify, but as a representative example to the systems stability and correctness of the algorithm we consider the step response of the system with the controller given by the center of mass calculation on the region and simulate the system using the MATLAB SIMULINK model of Figure 5.7. The stability of the system is verified in Figure 5.8.

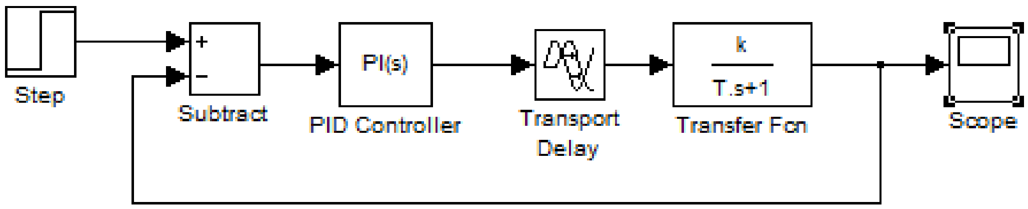

Figure 5.7: Simulation model for PI-control of first order systems with time delay.

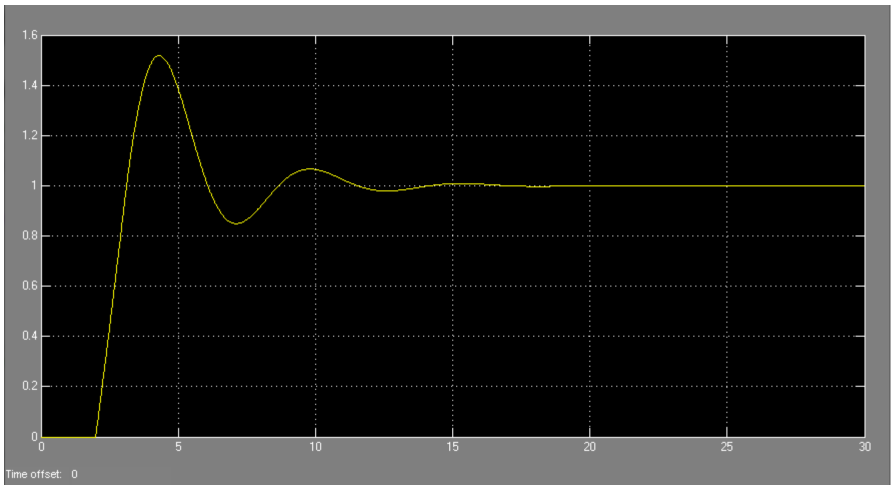

Figure 5.8: System response with $k_p = 3.347755$ and $k_i = 1.287736$.

5.4.5 PI-Control, Open-loop Unstable Plant

Let $k = 1$, $L = 1$ sec, and $T = -2$ sec. Running the [PI] algorithm for first order systems with time delay on the embedded system produces a narrowed k_p range of

$$k_p \in (-2.536559, -1.000000)\ .$$

Sweeping over this k_p range we achieve the stability region of Figure 5.9.

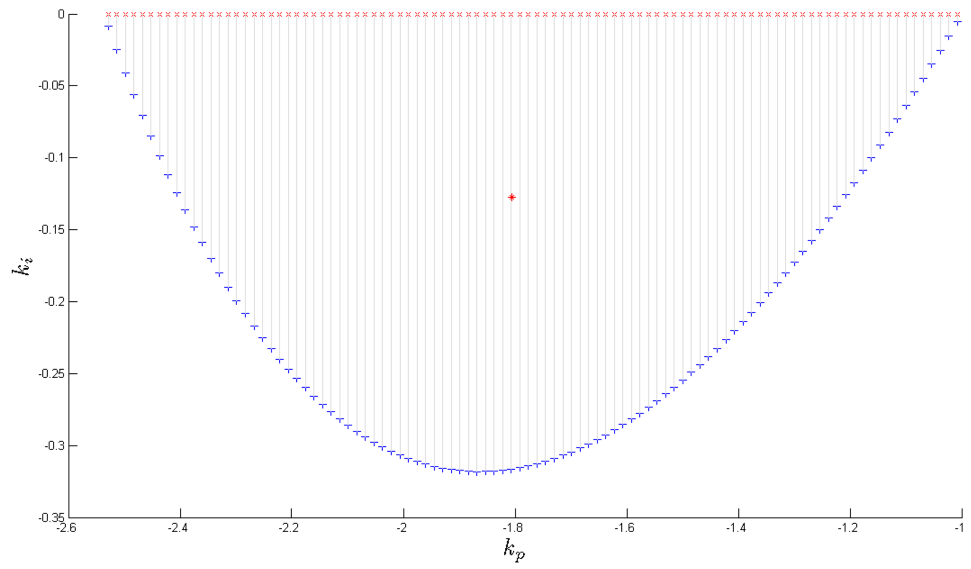

Figure 5.9: The stabilizing set of (k_p, k_i) values with "non-fragile" controller indicated by the star.

Calculating the center of mass of the 2D shape produces the "non-fragile" PI controller with

$$k_p = -1.806312$$

$$k_i = -0.127288\ .$$

We will consider a single point inside the stability region as a representative example to the systems stability and correctness of the algorithm because verification of the stability of the system over all possible (k_p, k_i) pairs would be impossible. We consider the step response of the system with the controller given by the center of mass calculation on the region and simulate the system using the MATLAB SIMULINK model of Figure 5.7. The stability of the system is verified in Figure 5.10.

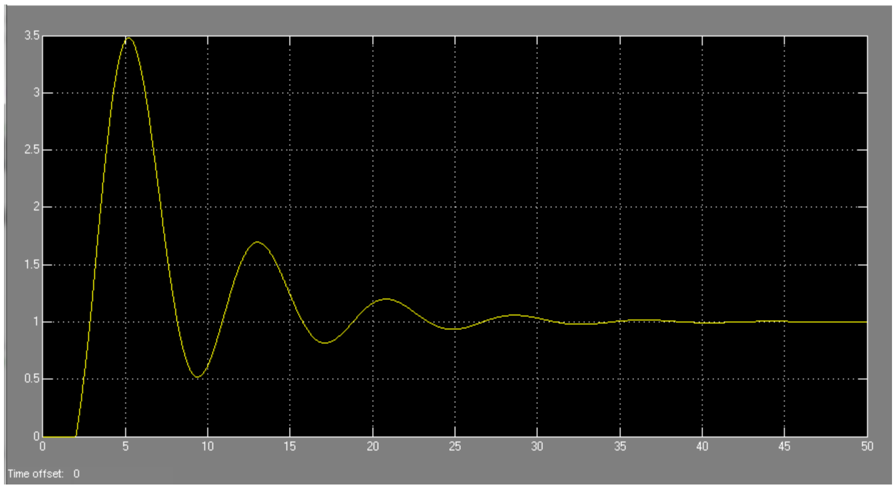

Figure 5.10: System response with $k_p = -1.806312$ and $k_i = -0.127288$.

5.5 Summary

In this chapter we extended the embedded systems implementation to first order systems with time delay applied to the characterization of stabilizing constant gain controllers, pure integral controllers, and proportional-integral controllers. Each case was tested against examples to verify correctness. The major sub algorithm developed was to find the intersection of two curves quickly and efficiently within some known interval of the real line. In the next chapter we will develop the computational algorithm needed to design constant gain controllers that achieve a desired damping.

6. CONSTANT GAIN STABILIZATION WITH DAMPING

Thus far, all of our polynomials have been polynomials with real coefficient. We have implemented algorithms for P, PI, and PID results that were based on generalizations of the Hermite-Biehler Theorem for both real valued polynomials and quasipolynomials [24]. The subject of designing a constant gain controller to achieve a desired degree of damping makes it necessary for us to consider polynomials with complex coefficients. An appropriate generalization of the Hermite-Biehler Theorem for complex polynomials is derived in [24] and used to develop a methodology to characterize all gain values to stabilize a given plant with a desired degree of damping. This chapter will discuss the practical implementation of such an algorithm onto an embedded systems architecture.

The algorithm covered here follows along similar lines as the [P] algorithm covered in chapter 2. The introduction of the complex coefficient polynomials requires an extension to the `Polynomial` class to handle complex arithmetic and other intricacies created by complex coefficients. To the best of our knowledge an extension of the stabilization with desired damping results to the case of PI or PID control is yet to be carried out.

6.1 Algorithm for Stabilization with Desired Damping

We present the systematic algorithm for constant gain stabilization with desired damping covered in [24] with only the necessary computational steps. Keep in mind that all of the terms presented in the algorithm below are taken from [24] and will be properly defined when the sub algorithms are developed.

1. Initialization

a) Set $n = max[deg(D(s)), deg(N(s))]$.

b) Set $m = deg(N(s))$.

2. Real-Imaginary Decompositions of N and D.

   - Determine $N(s_1 e^{j\phi})$ and $D(s_1 e^{j\phi})$.
   - Scale $N(s_1 e^{j\phi})$ and $D(s_1 e^{j\phi})$ by dividing each by the leading coefficient of $D(s_1 e^{j\phi})$.
   - Produce the Real-Imaginary decompositions N'_R, N'_I, D'_R, D'_I.

3. Determine $p_1(\omega)$, $p_2(\omega)$, $q(\omega)$, and $D^*(w)$.

4. Find and sort the roots of $q(\omega)$ and $D^*(\omega)$.

   a) Find the roots of $q(\omega)$ that are real, distinct, finite, and of odd multiplicity

   b) Sort the roots as $-\infty = \omega_0 < \omega_1 < \omega_2 < \ldots \omega_{m-1} < \omega_m = \infty$ where we define $\omega_0 = -\infty$ and $\omega_m = \infty$.

   c) Find the roots of $D^*(w)$.

      - Set $lrDs$ equal to the number of roots of $D^*(w)$ in the OLHP.
      - Set $rrDs$ equal to the number of roots of $D^*(w)$ in the ORHP.

5. Define the set of admissible strings, A.

6. Define the imaginary signature $\gamma(I)$.

7. Determine the set $F^* = \{I \in A | \gamma(I) = n + lrDs - rrDs\}$.

8. Apply Theorem 8.4.1 in [24] to get K_ϕ by following these steps:

a) Check if $G(s_1 e^{j\phi}) = \frac{N(s_1 e^{j\phi})}{D(s_1 e^{j\phi})}$ is Hurwitz stable by examining the roots of $D(s_1 e^{j\phi})$ and set $K_0 = \{0\}$ if $G(s_1 e^{j\phi})$ is Hurwitz stable and $K_0 = \emptyset$ otherwise.

b) For each $I_r \in F^*$ find K_r for $r = 1, 2, \ldots, s$.

c) if $i_{m+1} = -1$ then for $t = 0, 1, 2, \ldots, m$ let,

$$K_r = \left(\max_{i_t \in I_r, i_t sgn[p_2(\omega_t)]=1} \left[-\frac{p_1(\omega_t)}{p_2(\omega_t)} \right], \right. \\ \left. \min \left[0, \min_{i_t \in I_r, i_t sgn[p_2(\omega_t)]=-1} \left[-\frac{p_1(\omega_t)}{p_2(\omega_t)} \right] \right] \right)$$

d) if $i_{m+1} = 1$ then for $t = 0, 1, 2, \ldots, m$ let

$$K_r = \left(\max \left[0, \max_{i_t \in I_r, i_t sgn[p_2(\omega_t)]=1} \left[-\frac{p_1(\omega_t)}{p_2(\omega_t)} \right] \right], \right. \\ \left. \min \left[0, \min_{i_t \in I_r, i_t sgn[p_2(\omega_t)]=-1} \left[-\frac{p_1(\omega_t)}{p_2(\omega_t)} \right] \right] \right)$$

e) $K_\phi = \bigcup_{r=0}^{s} K_r$ where $s = card(F^*)$.

9. Calculate $N_\alpha(s) = N(s - \alpha)$ and $D_\alpha(s) = D(s - \alpha)$ and run the [P] algorithm for the plant $G_\alpha = \frac{N_\alpha}{D_\alpha}$ to obtain K_α.

10. Take the interval intersection $K = K_\alpha \cap K_\phi$.

Though the flow of the algorithm is very similar to the [P] algorithm, there are many differences in the underlying calculation that need careful attention. We will expand on these individually.

6.1.1 Real-Imaginary Decompositions of N and D

In [24] the real and imaginary decompositions refer to the fact that if the polynomial being decomposed is evaluated at $s_1 = j\omega$, then one of the components evaluates out to the real part while the other component evaluates out to the imaginary part. The first step is to produce the terms of $N(s_1 e^{j\phi})$ and $D(s_1 e^{j\phi})$ by replacing s with $s_1 e^{j\phi}$. This process introduces the need for a `ComplexPolynomial` class or an extension to the `Polynomial` class to handle complex coefficients. The C++ standard is well equipped with the libraries to do complex arithmetic through the `complex.h` header file. Therefore, a `ComplexPolynomial` class or an extension to the `Polynomial` class is straight forward to implement and we will proceed under the assumption that we are fully capable of handling complex polynomial arithmetic in the same fashion as real polynomial arithmetic.

For better understanding, let us consider an example:

Ex. 6 — Let $P(s) = s^2 + 2s - 2$. Determine the complex polynomial obtained by replacing s with $s_1 e^{j\phi}$ where $\phi = \frac{\pi}{6}$.

Answer (Ex. 6) —

$$\begin{aligned}
P(s_1 e^{j\phi}) &= (s_1 e^{j\phi})^2 + 2(s_1 e^{j\phi}) - 2 \\
&= (e^{j\phi})^2 s_1^2 + 2(e^{j\phi}) s_1 - 2 \\
&= [\cos(\phi) + j\sin(\phi)]^2 s_1^2 + 2[\cos(\phi) + j\sin(\phi)] s_1 - 2 \\
&= (0.5 + 0.866j) s_1^2 + (1.73 + j) s_1 - 2
\end{aligned}$$

Ex. 7 — Let $P(s) = s^3 + 3s^2 + 4s$. Determine the complex polynomial obtained by replacing s with $s_1 e^{j\phi}$ where $\phi = \frac{\pi}{6}$.

Answer (Ex. 7) —

$$\begin{aligned}
P(s_1 e^{j\phi}) &= (s_1 e^{j\phi})^3 + 3(s_1 e^{j\phi})^2 + 4(s_1 e^{j\phi}) \\
&= (e^{j\phi})^3 s_1^3 + 3(e^{j\phi})^2 s_1^2 + 4(e^{j\phi}) s_1 \\
&= [\cos(\phi) + j\sin(\phi)]^3 s_1^3 + 3[\cos(\phi) + j\sin(\phi)]^2 s_1^2 + 4[\cos(\phi) + j\sin(\phi)] s_1 \\
&= j s_1^3 + (1.5 + 2.6j) s_1^2 + (3.46 + 2j) s_1
\end{aligned}$$

Here it can be seen that the effective computational result is a multiplication of each original coefficient of N and D with the complex value $[\cos(\phi) + j\sin(\phi)]^i$ where i is the same power of the term that the original coefficient belonged to. Therefore we generate a special function inside our `Polynomial` class to handle this basic conversion and we return a complex polynomial value to be handled next.

Once we have the representations of $N(s_1 e^{j\phi})$ and $D(s_1 e^{j\phi})$ in our `Polynomial` object we then normalize both by dividing them by the leading coefficient of $D(s_1 e^{j\phi})$ so that the leading coefficient of $D(s_1 e^{j\phi})$ is now one. The normalized polynomials are now labeled as $N'(s_1)$ and $D'(s_1)$. Now the decomposition into real and imaginary parts for $N'(s_1)$ and $D'(s_1)$ is ready to be executed. But first, for completeness of understanding we will restate some definitions from [24]. We have

$$\begin{aligned}
N(s_1 e^{j\phi}) &= (a_m + jb_m) s_1^m + (a_{m-1} + jb_{m-1}) s_1^{m-1} + \dots \\
&\qquad + (a_1 + jb_1) s_1 + (a_0 + jb_0), \qquad a_m + jb_m \neq 0 \\
D(s_1 e^{j\phi}) &= (c_n + jd_n) s_1^n + (c_{n-1} + jd_{n-1}) s_1^{n-1} + \dots \\
&\qquad + (c_1 + jd_1) s_1 + (c_0 + jd_0), \qquad c_n + jd_n \neq 0
\end{aligned}$$

where $n \geq m$. And,

$$\begin{aligned}
N'(s_1) &\triangleq \frac{1}{c_n + jd_n} N(s_1 e^{j\phi}) \\
&= (a'_m + jb'_m)s_1^m + (a'_{m-1} + jb'_{m-1})s_1^{m-1} + \cdots + (a'_1 + jb'_1)s_1 + (a'_0 + jb'_0) \\
D'(s_1) &\triangleq \frac{1}{c_n + jd_n} D(s_1 e^{j\phi}) \\
&= s_1^n + (c'_{n-1} + jd'_{n-1})s_1^{n-1} + \cdots + (c'_1 + jd'_1)s_1 + (c'_0 + jd'_0)
\end{aligned}$$

where

$$\begin{aligned}
a'_i &= \frac{a_i c_n + b_i d_n}{c_n^2 + d_n^2} \\
b'_i &= \frac{-a_i d_n + b_i c_n}{c_n^2 + d_n^2}, i = 0, 1, \ldots, m \\
c'_i &= \frac{c_i c_n + d_i d_n}{c_n^2 + d_n^2} \\
d'_i &= \frac{-c_i d_n + d_i c_n}{c_n^2 + d_n^2}, i = 0, 1, \ldots, n-1 \ .
\end{aligned}$$

Now the real-imaginary decomposition follows directly as

$$\begin{aligned}
N'(s_1) &= N'_R(s_1) + N'_I(s_1) \\
D'(s_1) &= D'_R(s_1) + D'_I(s_1)
\end{aligned}$$

where

$$\begin{aligned}
N'_R(s_1) &= a'_0 + jb'_1 s_1 + a'_2 s_1^2 + jb'_3 s_1^3 + \dots \\
N'_I(s_1) &= jb'_0 + a'_1 s_1 + jb'_2 s_1^2 + a'_3 s_1^3 + \dots \\
D'_R(s_1) &= c'_0 + jd'_1 s_1 + c'_2 s_1^2 + jd'_3 s_1^3 + \dots \\
D'_I(s_1) &= jd'_0 + c'_1 s_1 + jd'_2 s_1^2 + c'_3 s_1^3 + \dots
\end{aligned}$$

Because all these terms are well defined after the computation of $N(s_1 e^{j\phi})$ and $D(s_1 e^{j\phi})$ the computation of $N'_R(s_1), N'_I(s_1), D'_R(s_1), D'_I(s_1)$ is straight forward given the above definitions.

6.1.2 Determining $p_1(\omega)$, $p_2(\omega)$, $q(\omega)$, and $D^*(w)$

In a similar fashion as in the [P] algorithm we will have to replace s_1 with $j\omega$ for $N'_R(s_1)$, $N'_I(s_1)$, $D'_R(s_1)$ and $D'_I(s_1)$. Once this conversion is complete, [24] defines $p_1(\omega)$, $p_2(\omega)$, $q(\omega)$, and $D^*(w)$ as follows

$$\begin{aligned}
p_1(\omega) &= {D'_R}^2(j\omega) - {D'_I}^2(j\omega) \\
p_2(\omega) &= D'_R(j\omega)N'_R(j\omega) - D'_I(j\omega)N'_I(j\omega) \\
q(\omega) &= \frac{1}{j}[D'_R(j\omega)N'_I(j\omega) - D'_I(j\omega)N'_R(j\omega)] \\
D^*(\omega) &= D'_R(j\omega) - D'_I(j\omega)\ .
\end{aligned}$$

Unlike the first `stojw()` function from the [P] algorithm where we worked with real coefficient polynomials, now our coefficients are complex. There is no special treatment for the sign of j^e where e was an even integer. Because we are working with complex data types in the C++ standard, we can directly apply the complex mathematics needed to carry out such a conversion. However, we do find a small discrepancy between the definitions of $p_1(\omega)$, $p_2(\omega)$ and $q(\omega)$ and that is in the factor of $\frac{1}{j}$ found in $q(\omega)$. We can account for this in our `stojw()` function by considering what the effect of $\frac{1}{j}$ is on such a conversion. The computational net result is that every term $(j\omega)^i$ effectively becomes $(j)^{i-1}\omega^i$. With this in mind the `stojw()` algorithm is detailed in Algorithm 9.

Algorithm 9 stojw(k)

Precondition: $P(s) = c_n s^n + c_{n-1}s^{n-1} + \cdots + c_a s + c_0$ is the complex valued polynomial being operated on.

```
function STOJW($k$)                              ▷ $k = 0$ for $p_1$ and $p_2$, $k = 1$ for $q$
    for $i = 0 \to deg(P(s)) - 1$ do
        $c_i \leftarrow c_i j^{i-k}$
```

After the $j\omega$ conversion, the three polynomials $p_1(\omega)$, $p_2(\omega)$ and $q(\omega)$ become polynomials with real coefficients. This follows from the fact that the real decomposition is purely real and the imaginary decomposition is purely imaginary. In the definitions of $p_1(\omega)$ and $p_2(\omega)$ the purely imaginary decompositions are always multiplied by another purely imaginary decomposition, which in turn produces a purely real polynomial. In the definition of $q(\omega)$ this is not the case, however, the $\frac{1}{j}$ term removes the purely imaginary component, leaving only a purely real polynomial. From this point forward the main part of the algorithm no longer needs to handle complex polynomials, except for a small check involving $D^*(\omega)$.

6.1.3 Complex Root Finding for $D^*(\omega)$

For $D^*(\omega)$ we still have a complex coefficient polynomial, even after the $s \to j\omega$ conversion. For our real coefficient polynomials we used [26] to find the roots of a polynomial. Similarly, the same algorithm exists for complex coefficient polynomials and was found in [47], which is also based on the Jenkins-Traub algorithm. The roots of $D^*(\omega)$ are needed in order to properly define the set of strings A, similar to the [P] algorithm in which $N^*(s)$ was used. This creates the $lrDs$ and $rrDs$ values needed in step 7 to define the set F^*.

6.1.4 Defining the Set of Strings, A

In [24] definition 8.4.1 states that A is the set of all possible strings of 1's, 0's, and -1's of length $m+2$ subject to the following restrictions. For every $I = \{i_0, i_1, \ldots, i_m, i_{m+1}\} \in A, i_{m+1}$ is either -1 or +1. For all other $t = 0, \ldots, m, i_t$ is equal to 0 if the corresponding $j\omega_t$ is a $j\omega$ axis root of $D^*(s_1)$; if $p_2(\omega_t) = 0$ but $p_1(\omega_t) \neq 0$ for some $t = 0, 1, 2, \ldots, m$, then $i_t = sgn[p_1(\omega_t)]$; and finally i_t is equal to -1 or +1 if neither of these conditions is satisfied.

A key implementation detail is determining if the corresponding $j\omega_t$ is a $j\omega$ axis root of $D^*(s_1)$. We saw previously that we created the complex polynomial $D^*(\omega)$. However, if we consider that $D^*(s_1)|_{j\omega_t} = D^*(\omega)|_{\omega_t}$ then checking this condition is reduced to a simple polynomial evaluation of the root, ω_t, of $q(\omega)$. Polynomial evaluation and root finding of $q(\omega)$, a real coefficient polynomial, has already been covered. Lastly, we must consider the case of $p_2(\omega_t) = 0$ but $p_1(\omega_t) \neq 0$. But again, $p_1(\omega_t)$ and $p_2(\omega_t)$ are both real coefficient polynomials and this check requires a simple polynomial evaluation.

6.1.5 Determining K_ϕ

Steps 6, 7, and 8 all work together to determine the interval K_ϕ. Step 6 is implemented through definition 8.4.2 of [24]. This is given to us as

$$\gamma(I) = \frac{1}{2}\left\{ i_0(-1)^{m-1} + 2\sum_{r=1}^{m-1} i_r(-1)^{m-1-r} - i_m \right\} sgn[i_{m+1}q(\infty)] \ .$$

It may be easier for implementation purposes to consider this as the expression of a dot product, that is, we separate out the coefficients from the i terms, keeping in mind that $sgn(xy) = sgn(x)sgn(y)$. To this end, let us define $\bar{\gamma}$ as the coefficient vector $[\frac{(-1)^{m-1}}{2}, \frac{(-1)^{m-2}}{2}, \ldots, \frac{(-1)^{1}}{2}, \frac{(-1)^{0}}{2}, \frac{-1}{2}] \cdot sqn[q(\infty)]$ and $\bar{I}$ as the string vector $[i_0, i_1, \ldots, i_m] \cdot sgn[i_{m+1}]$. Then we can define $\gamma(I) = \bar{\gamma}\bar{I}^T$. Now computationally we can define $F^* = \{I \in A | \bar{\gamma}\bar{I}^T = n + lrDs - rrDs\}$, in step 7.

In step 8 we are applying Theorem 8.4.1 in [24] by examining each string in F^* and evaluating $-\frac{p_1(\omega_t)}{p_2(\omega_t)}$ and $sgn[p_2(\omega_t)]$ and performing a search over these values and their associated i_t value in the string. The search is constructing the K_r intervals. Once every string in F^* is considered and all K_r intervals found we take the intersection of these intervals to define K_ϕ

6.1.6 Determining K_α

In step 9 we are required to consider two new polynomials $N_\alpha(s) = N(s - \alpha)$ and $D_\alpha(s) = D(s - \alpha)$, then run the [P] algorithm with $N_\alpha(s)$ and $D_\alpha(s)$. Because the $N_\alpha(s)$ and $D_\alpha(s)$ are polynomials of real coefficients and the [P] algorithm is already formulated, the only computational challenge is to determine $N_\alpha(s)$ and $D_\alpha(s)$. Because of the utility of our `Polynomial` class, this is not difficult. We follow the steps of Algorithm 10.

Algorithm 10 POLYSHIFT(α)

Precondition: $P(s) = c_n s^n + c_{n-1}s^{n-1} + \cdots + c_a s + c_0$ is the polynomial being operated on

```
function POLYSHIFT($\alpha$)
    $T \leftarrow s - \alpha$
    $P \leftarrow 0$
    $d \leftarrow deg(P(s))$
    for $i = deg(P(s)) \rightarrow 2$ step $-1$ do
        $M \leftarrow T$
        for $j = 1 \rightarrow d - 1$ do
            $M \leftarrow M \cdot T$
        $P \leftarrow P + c_i \cdot M$
        $d \leftarrow d - 1$
    $P \leftarrow P + c_1 T + c_0$
```

In this algorithm we see the value of the `Polynomial` class allowing us easy addition and multiplication of polynomials.

Once the [P] algorithm is run on the plant defined as $\frac{N_\alpha(s)}{D_\alpha(s)}$, it produces the interval K_α and our final result is produced by using our interval intersection algorithm to find $K = K_\alpha \cap K_\phi$.

6.2 Examples and Results

Here we consider the example problem given in [24] for constant gain stabilization with desired damping. Let the plant be

$$N(s) = s^2 + 2s - 2$$
$$D(s) = s^3 + 3s^2 + 4s$$

and let the parameters be

$$\alpha = 0.5$$
$$\phi = \frac{\pi}{6} \ .$$

When executing this algorithm for this specific plant on our embedded system we produce the following results:

$$\begin{aligned} p_1(\omega) &= 1.0\omega^6 - 3.0\omega^5 + 5.0\omega^4 - 12.0\omega^3 + 16.0\omega^2 \\ p_2(\omega) &= -0.5\omega^5 + 2.0\omega^4 - 7.0\omega^3 + 11.0\omega^2 + 4.0\omega \\ q(\omega) &= -0.866025\omega^5 + 1.732051\omega^4 - 1.732051\omega^3 - 5.196152\omega^2 + 6.928203\omega \ . \end{aligned}$$

The set F^* is

$$\left\{ \begin{array}{c} (-1,-1,0,-1,-1,1) \\ (-1,-1,0,1,-1,-1) \\ (-1,1,0,-1,-1,-1) \\ (1,-1,0,1,1,1) \\ (1,1,0,-1,1,1) \\ (1,1,0,1,1,-1) \end{array} \right\}$$

and

$$K_\phi = (-0.725001, 0.000000)$$
$$K_\alpha = (-0.763932, -0.500000)$$

with the final result being that the plant is stabilized and all zeros bound by the specified region for every

$$k_p \in (-0.725001, -0.500000)\ .$$

The stability of the system over all possible k_p values is impossible to verify, but as a representative example to the systems stability and correctness of the algorithm we consider $k_p = -0.625$ and use MATLAB's SIMULINK to generate the step response. Using the system of Figure 2.1 we obtain the step response presented in Figure 6.1.

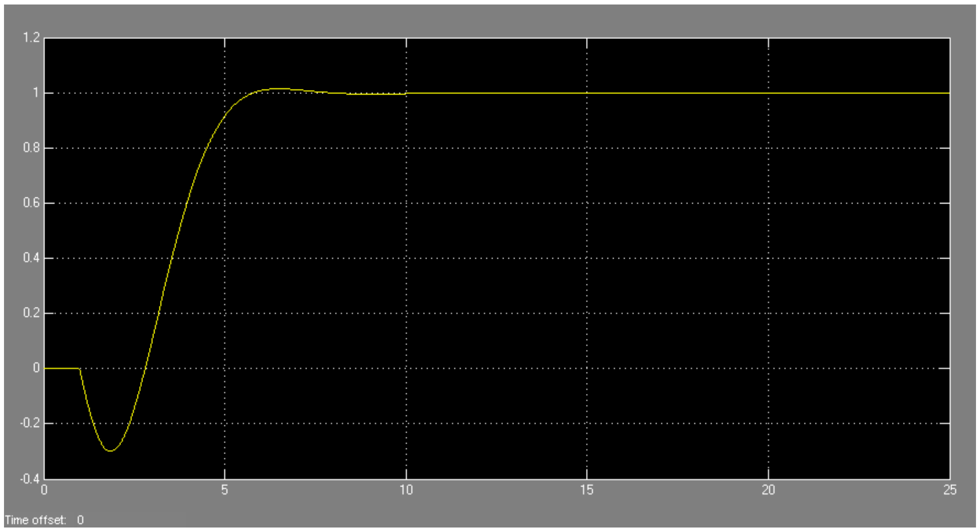

Figure 6.1: System response with $k_p = -0.625$.

Further we can examine the pole/zero plots of the system in this k_p range and verify that the zeros are placed in the region specified by the parameters ϕ and α. A depiction of this can be shown in Figure 6.2.

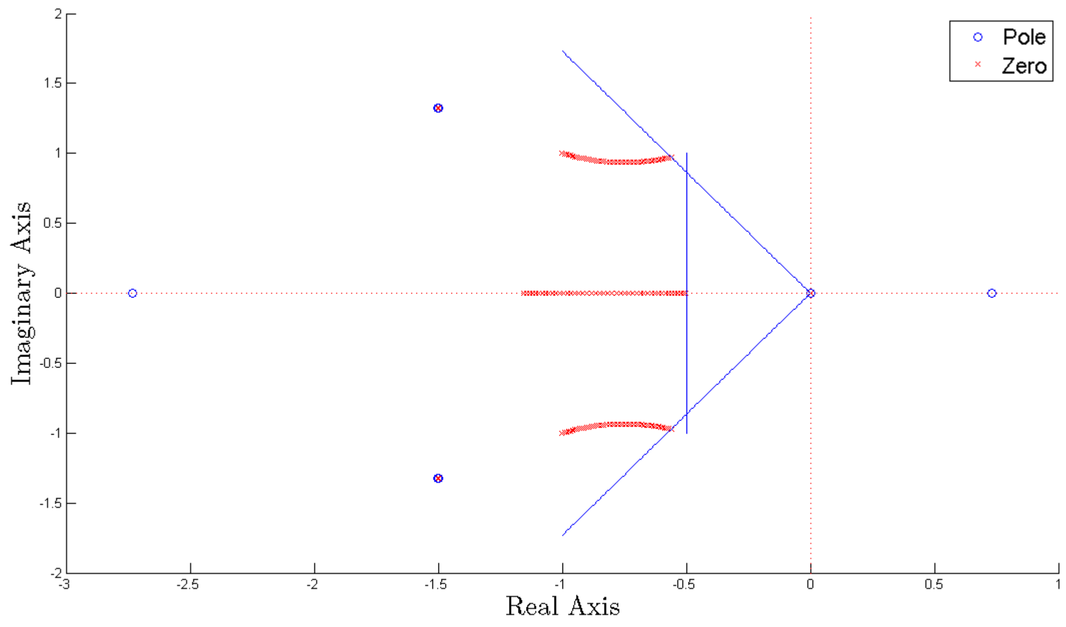

Figure 6.2: Plot of the pole/zero locations for $k_p \in (-0.725, -0.500)$.

This concludes the verification of the example provided in [24]. Now let us consider a different example in this same context. Consider the plant

$$N(s) = s^3 + 2s^2 - 2s + 7$$

$$D(s) = s^4 + 5s^3 + 3s^2 + 4s$$

and let the parameters be $\alpha = 0.1$ and $\phi = \frac{\pi}{32}$. When executing this algorithm for this specific plant on our embedded system we find the following results:

$$p_1(\omega) = 1.0\omega^8 - 0.9802\omega^7 + 19.115\omega^6 - 0.6182\omega^5 - 30.231\omega^4 - 2.352\omega^3 + 16.0\omega^2$$
$$p_2(\omega) = -0.0980\omega^7 + 3.038\omega^6 - 1.855\omega^5 + 18.352\omega^4 + 9.964\omega^3 - 28.596\omega^2 - 2.744\omega$$
$$q(\omega) = -0.9952\omega^7 + 0.3902\omega^6 - 8.88\omega^5 - 5.41\omega^4 - 47.426\omega^3 + 4.097\omega^2 - 27.865\omega \ .$$

The set F^* is

$$\begin{Bmatrix} (-1,-1,-1,0,-1,-1,-1,-1) & (-1,-1,-1,0,1,-1,-1,1) \\ (-1,-1,-1,0,1,1,-1,-1) & (-1,-1,1,0,-1,-1,-1,1) \\ (-1,-1,1,0,-1,1,-1,-1) & (-1,-1,1,0,1,1,-1,1) \\ (-1,1,-1,0,1,-1,-1,-1) & (-1,1,1,0,-1,-1,-1,-1) \\ (-1,1,1,0,1,-1,-1,1) & (-1,1,1,0,1,1,-1,-1) \\ (1,-1,-1,0,-1,-1,1,1) & (1,-1,-1,0,-1,1,1,-1) \\ (1,-1,-1,0,1,1,1,1) & (1,-1,1,0,-1,1,1,1) \\ (1,1,-1,0,-1,-1,1,-1) & (1,1,-1,0,1,-1,1,1) \\ (1,1,-1,0,1,1,1,-1) & (1,1,1,0,-1,-1,1,1) \\ (1,1,1,0,-1,1,1,-1) & (1,1,1,0,1,1,1,1) \end{Bmatrix}$$

and

$$K_\phi = (0.000000, 0.179735)$$
$$K_\alpha = (0.0519324006095, 0.159385) \ .$$

with the final result being that the plant is stabilized and all zeros bound by the specified region for every

$$k_p \in (0.051932, 0.159385) \ .$$

Examining the pole/zero plots of the system in this k_p range we verify that the zeros are placed in the region specified by the parameters ϕ and α. A depiction of this result is shown in Figure 6.3.

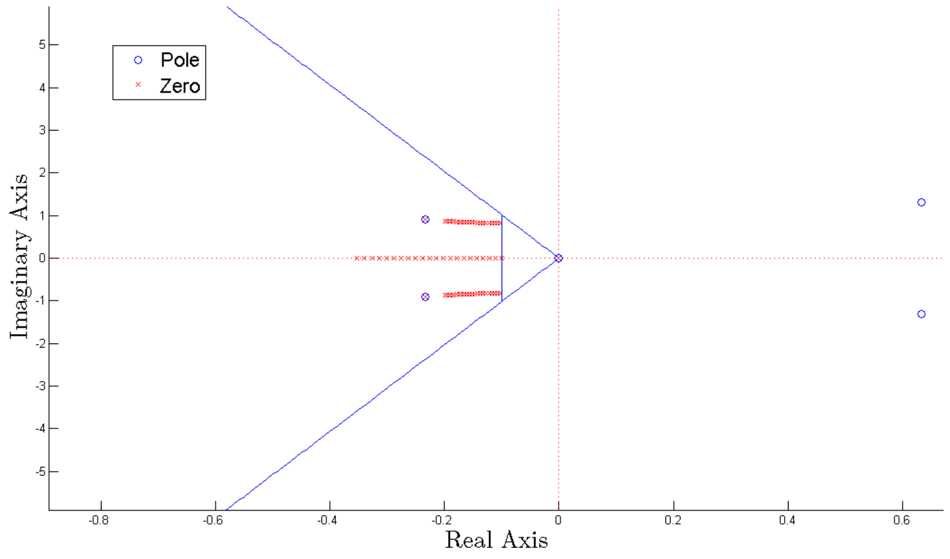

Figure 6.3: Plot of the pole/zero locations for $k_p \in (0.051932, 0.159385)$.

6.3 Summary

In this chapter we modified our `Polynomial` class to accept polynomials with complex coefficient. This required updating the polynomial arithmetic functions of the class to do complex arithmetic on the coefficients and was necessary to successfully extend the constant gain stabilization algorithm from chapter 2 to handle stabilization with a desired damping. The new functionality was tested and verified against examples. In the next and last chapter we show extension of the embedded systems algorithm for constant gain stabilization to the case of discrete time plants.

7. CONSTANT GAIN STABILIZATION FOR DISCRETE TIME PLANTS

In this chapter we explore the embedded systems implementation of constant gain stabilization for the discrete time case. Until now, all of our systems have been assumed to be continuous time systems. The results require a derivation of the generalized Hermite-Biehler theorem for Schur polynomials. Schur polynomials are a special class of symmetric polynomials and their treatment in control theory arises from the analysis of discrete time linear systems. A Schur polynomial is a stable polynomial for discrete time linear systems, that is, a Schur polynomial has all its roots in the open unit disc. The algorithm for a complete characterization of all constant gain stabilizing controllers is drawn from [24], however the results for PI and PID stabilization have not been completed.

7.1 Algorithm for Stabilization For Discrete Time Plants

We present the systematic algorithm for constant gain stabilization for discrete time plants covered in [24] with only the necessary computational steps. We will refer to this algorithm as the [P] algorithm for discrete time plants.

1. Initialization

   a) Set $n = deg(D(z))$ and $m = deg(N(z))$.

   b) Determine the zeroes of $N(z)$ that exist inside the open unit disc.

   c) Set $zNz = card(\{z_i | N(z_i) = 0, \|z_i\| < 1\})$.

   d) Set pNz equal to the number of poles of $N(z)$ in the open unit disc.

2. Determine the real-imaginary decompositions of $H_r(z) = p_1(z)$, $p_2(z)$, and $H_i(z) = jq(z)$.

a) $H(z) = D(z)N(\frac{1}{z}) = H_r(z) + H_i(z)$.

b) $H_r(z) = \frac{1}{2}[D(z)N(\frac{1}{z}) + D(\frac{1}{z})N(z)]$.

c) $H_i(z) = \frac{1}{2}[D(z)N(\frac{1}{z}) - D(\frac{1}{z})N(z)]$.

d) $p_2(z) = N(z)N(\frac{1}{z})$.

3. Find the distinct upper half plane unit circle zeroes of $H_i(z)$ with odd multiplicity at the corresponding ω_t argument and sort them according to $0 \leq \omega_0 < \omega_1 < \cdots < \omega_l \leq \pi$ where ω_t are the real roots of $q(\omega) = \frac{1}{j}H_i(z)|_{z=e^{j\omega}}$.

4. Generate the set A of all possible strings of $\{-1, 1\}$ of length l.

5. Determine the imaginary signature $\gamma(I)$.

6. Determine the set $F^* = \{I \in A|\gamma(I) = n - (zNz - pNz)\}$.

7. For each $I_r \in F^*$ Apply Theorem 9.5.1 of [24] by finding,

$$K_r = \left(\max_{i_t \in I_r, i_t = 1} \left[-\frac{p_1(\omega_t)}{p_2(\omega_t)} \right], \min_{i_t \in I_r, i_t = -1} \left[-\frac{p_1(\omega_t)}{p_2(\omega_t)} \right] \right)$$

with $K_p = \cup_{r=1}^{s} K_r$ where $s = card(F^*)$.

Though the flow of the algorithm is very similar to the [P] algorithm from chapter 2, there are many differences in the underlying calculation that need careful attention. We will expand on these individually.

7.1.1 Real-Imaginary Decompositions of $N(z)$ and $D(z)$

We saw previously that

$$\begin{aligned}
H(z) &= D(z)N(\frac{1}{z}) = H_r(z) + H_i(z) \\
H_r(z) &= \frac{1}{2}[D(z)N(\frac{1}{z}) + D(\frac{1}{z})N(z)] \\
H_i(z) &= \frac{1}{2}[D(z)N(\frac{1}{z}) - D(\frac{1}{z})N(z)] \\
p_2(z) &= N(z)N(\frac{1}{z}) \ .
\end{aligned}$$

Let us first examine an example to see how such a decomposition will look computationally.

Ex. 8 — Find $H_r(z)$ and $H_i(z)$ given that $N(z) = 100z^3 + 2z^2 + 3z + 11$ and $D(z) = 100z^5 + 2z^4 + 5z^3 - 41z^2 + 52z + 70$.

Answer (Ex. 8) — Begin by first determining that

$$\begin{aligned}
N(\frac{1}{z}) &= z^{-3}(11z^3 + 3z^2 + 2z + 100) \\
D(\frac{1}{z}) &= z^{-5}(70z^5 + 52z^4 - 41z^3 + 5z^2 + 2z + 100) \ .
\end{aligned}$$

Now,

$$\begin{aligned}
D(z)N(\frac{1}{z}) &= (100z^5 + 2z^4 + 5z^3 - 41z^2 + 52z + 70)(11z^3 + 3z^2 + 2z + 100)z^{-3} \\
D(\frac{1}{z})N(z) &= z^{-5}(70z^5 + 52z^4 - 41z^3 + 5z^2 + 2z + 100)(100z^3 + 2z^2 + 3z + 11) \ .
\end{aligned}$$

And put it all together with

$$\begin{aligned} H_r(z) &= \frac{1}{2}[D(z)N(\frac{1}{z}) + D(\frac{1}{z})N(z)] \\ H_i(z) &= \frac{1}{2}[D(z)N(\frac{1}{z}) - D(\frac{1}{z})N(z)] \ . \end{aligned}$$

Finally, denote $N_f(z) = 11z^3 + 3z^2 + 2z + 100$ and $D_f(z) = 70z^5 + 52z^4 - 41z^3 + 5z^2 + 2z + 100$, then

$$\begin{aligned} D(z)N(\frac{1}{z}) \pm D(\frac{1}{z})N(z) &= D(z)N_f(z)z^{-3} \pm D_f(z)N(z)z^{-5} \\ &= [D(z)N_f(z)z^2 \pm D_f(z)N(z)]z^{-5} \\ &= [D(z)N_f(z)z^{n-m} \pm D_f(z)N(z)]z^{-n} \ . \end{aligned}$$

We see that an important step in formulating these decompositions is to produce the term $N(\frac{1}{z})$ and $D(\frac{1}{z})$. But through the above example we see that this is equivalent to a computational flipping of the coefficient vector for $N(z)$ and $D(z)$ and multiplication by a z^{-d} where d is the degree of the polynomial. Thus, a simple function can be added to the `Polynomial` class to flip the coefficient vector of a `Polynomial` object and return the new polynomial. Lastly, we must determine how to properly work with the z^{-d} term. In the above example we see that

$$D(z)N(\frac{1}{z}) \pm D(\frac{1}{z})N(z) = [D(z)N_f(z)z^{n-m} \pm D_f(z)N(z)]z^{-n} \ .$$

Since $n \geq m$, because the plant is a proper transfer function, $n - m \geq 0$, and we have a polynomial expression that we know how to computationally handle with our `Polynomial` class, except for the z^{-n} term. If we consider the full expressions of real-imaginary decompositions as they fit into the entire algorithm we have:

$$\begin{aligned}
p_1(z) &= H_r(z) = \frac{1}{2}[D(z)N(\frac{1}{z}) + D(\frac{1}{z})N(z)] = [D(z)N_f(z)z^{n-m} + D_f(z)N(z)]z^{-n} \\
jq(z) &= H_i(z) = \frac{1}{2}[D(z)N(\frac{1}{z}) - D(\frac{1}{z})N(z)] = [D(z)N_f(z)z^{n-m} - D_f(z)N(z)]z^{-n} \\
p_2(z) &= N(z)N(\frac{1}{z}) = N(z)N_f(z)z^{-m} \ .
\end{aligned}$$

In step 3 of the [P] algorithm for discrete time plants we are required to find the distinct upper half plane unit circle zeroes of $H_i(z)$ with odd multiplicity. Yet, the additional z^{-n} term will not affect the zeros in the expression for $H_i(z)$. Therefore, it is not necessary for the computation of the zeros and can be ignored. Further, Steps 4 and 6 of this algorithm are not needed for any computations on the polynomials produced here. Only in Step 5 and 7 will the full polynomial expression be necessary. Details of these steps will be discussed later, however, it should be noted that the term, z^{-n}, is fully known at all times during the algorithm. Thus, any need of a polynomial evaluation can handle this term separately and there is no need to include it into the `Polynomial` object or incorporate extra utility to handle negative exponent polynomial terms.

We have shown that by ignoring the z^{-n} and handling it separately we can still meet the computational needs of the algorithm and maintain use of the utility of the `Polynomial` class when formulating the real-imaginary decomposition in the case of discrete time plants.

7.1.2 Roots of $H_i(z)$ and the Corresponding ω_t Arguments

In [24], part of the algorithm specifies a conversion to the frequency domain by setting $z = e^{j\omega}$. Even though such a conversion still only produces real polynomials, it does add more computation to the algorithm. If we closely examine why this conversion is needed we find out that the necessary ω_t values, which are the roots of $q(\omega)$, can be determined directly from the roots of $H_i(z)$. The corresponding arguments of the roots z_t are the roots ω_t. The ω_t values are needed to properly sort the roots generated in Step 3.

If we consider that $z_t = cos(\omega_t) + jsin(\omega_t) = e^{j\omega_t}$, then we may forgo the conversion to the frequency domain. Because the range of $y = \cos^{-1}(x)$ is $0 \leq y \leq \pi$ and the theory dictates that $0 \leq \omega_t \leq \pi$, we need only consider the real part of the root when finding ω_t. That is $\omega_t = \cos^{-1}(real(z_t))$. This provides to us our ω_t values needed for sorting of the roots without the need of converting to the frequency domain, saving on computation time.

7.1.3 Handling the Extra z^{-n} Term

As previously discussed, in steps 5 and 7, we require the full polynomial expressions of the real-imaginary decompositions, but we ignored the z^{-n} term when computing and storing these expressions. Now it is necessary to explain how to properly utilize this ignored term when executing the steps of this algorithm computationally.

In step 5 we are asked to compute the imaginary signature. This calculation requires the calculation of $sgn[q(\omega_0^+)]$. Because we avoided the frequency domain we consider the z domain equivalent, $sgn[\frac{1}{j}H_i(z_0^+)]$. However, $H_i(z)$ is a symmetric polynomial, and has a trivial root at $z = 1$. The corresponding argument for $z = 1$ is $\omega = 0$ because $0 = \cos^{-1}(1)$. Thus $\omega_0 = 0$ and $z_0 = 1$. This is true for any $H_i(z)$. Evaluation of $H_i(z_0^+)$ means we will also evaluate $z^{-n}|_{z_0^+}$, but since $z_0 = 1$, then $z^{-n}|_{z_0^+} \approx 1$ and the term doesn't effect the calculation of the $sgn[\frac{1}{j}H_i(z_0^+)]$. Thus it can again be ignored.

In step 7, however, the z^{-n} term cannot be ignored. We must evaluate the term $-\frac{p_1(\omega_t)}{p_2(\omega_t)}$. Since we skipped the frequency domain calculation we defer to the z-domain.

We consider $-\frac{p_1(z_t)}{p_2(z_t)}$ and recall that

$$p_1(z) = [D(z)N_f(z)z^{n-m} + D_f(z)N(z)]z^{-n}$$
$$p_2(z) = N(z)N_f(z)z^{-m} \ .$$

It then follows that

$$\begin{aligned} -\frac{p_1(z_t)}{p_2(z_t)} &= -\frac{[D(z_t)N_f(z_t)z_t^{n-m} + D_f(z_t)N(z_t)]z_t^{-n}}{N(z_t)N_f(z_t)z_t^{-m}} \\ &= -\frac{[D(z_t)N_f(z_t)z_t^{n-m} + D_f(z_t)N(z_t)]}{N(z_t)N_f(z_t)} \cdot \frac{z_t^{-n}}{z_t^{-m}} \\ &= -\frac{[D(z_t)N_f(z_t)z_t^{n-m} + D_f(z_t)N(z_t)]}{N(z_t)N_f(z_t)} \cdot \frac{z_t^{m}}{z_t^{n}} \ . \end{aligned}$$

Now the need for a negative exponent has been eliminated in expressing this evaluation. Thus, in step 7 we create temporary z^m and z^n `Polynomials` and use them in the evaluation of $-\frac{p_1(z_t)}{p_2(z_t)}$.

We have seen that by deferring the immediate need in step 2 to deal with a negative exponent termed polynomial, we have created a more efficient computational algorithm to characterize all constant gain stabilizing controllers for discrete time plants.

7.2 Examples and Results

In this section we consider the example problem given in [24] for constant gain stabilization of discrete time plants. Let

$$N(z) = 100z^3 + 2z^2 + 3z + 11$$
$$D(z) = 100z^5 + 2z^4 + 5z^3 - 41z^2 + 52z + 70 \ .$$

After executing this algorithm for this specific plant on our embedded system we find the following results.

The distinct upper half plane unit circle zeros of $H_i(z)$ with odd multiplicity, after sorting based on the ω_t argument, are are given as

$$z_0 = 1.000000 - j0.000000$$
$$z_1 = 0.861315 + j0.508071$$
$$z_2 = -0.486046 + j0.873933$$
$$z_3 = -1.000000 + j0.000000$$

The set F^* is

$$\begin{Bmatrix} (-1, 1, -1, -1) \\ (1, 1, -1, 1) \end{Bmatrix}$$

and the final result is that the plant is stabilized for every

$$k_p \in (-0.417762, -0.126272) \ .$$

It is impossible to verify the stability of the system over all possible k_p values, but as a representative example to the systems stability and correctness of the algorithm we consider $k_p = -0.2$ and use MATLAB's SIMULINK to generate the step response of the system. Using the system of Figure 7.1 we obtain the step response presented in Figure 7.2.

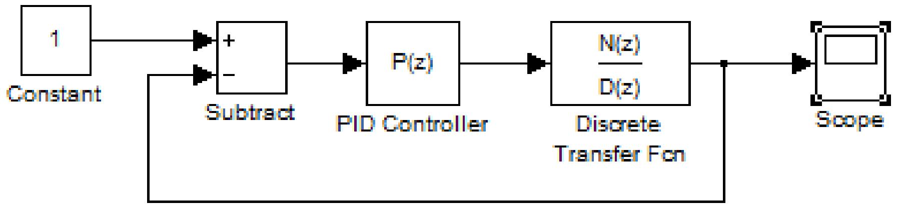

Figure 7.1: Simulation model for P-control of discrete time system.

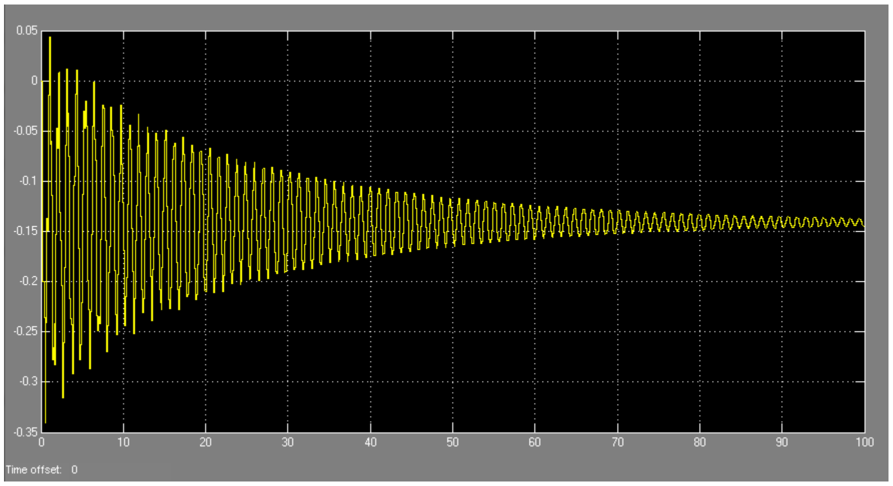

Figure 7.2: System response with $k_p = -0.2$.

7.3 Summary

In this chapter we successfully extended the constant gain stabilization algorithm of chapter 2 to the case of discrete time plants. This required a proper computation handling of Schur polynomials and polynomial arithmetic in the z-domain. Though, some of the original algorithm detailed in [24] had computations carried out in the frequency domain, we have shown with careful algebraic treatment all arithmetic can remain in the z-domain. Keeping the computations in the z-domain unifies the algorithm and makes it more efficient to carry out. The computational algorithm developed here was tested against an example to verify correctness.

8. CONCLUSIONS

We have presented a full embedded systems implementation of the results presented in [24] and explored the applicability of the theory to real time optimal auto-tuning for PID controllers. All examples found in [24] have been executed and verified on a TMDSEVM6678LE evaluation module of the TMS320C6678 DSP processor by Texas Instruments. A complete C++ library for use in this particular embedded system was produced to test the examples. We have found the methods presented in [24] to be fully capable of an embedded systems implementation after advanced search techniques were developed and optimizations made. These algorithms can be programmed on to an embedded controller design to optimally auto-tune PID controllers in real time with knowledge of the plant model.

Some difficulty to this application are computation time and knowledge of the plant model. In chapter 3 a brute force search over the parameter space was performed to find optimal controllers based on a given performance criteria. However, the amount of physical time needed to compute the performance metrics over the entire space of stable controllers was on the order of hours. To amend this problem, faster search algorithms were presented. The Coarse Grained to Fine Grained Search and Gradient Descent Search along with multi-processor parallel algorithms and hybrid search methods were discussed. A complete implementation of the Coarse Grained to Fine Grained Search and Gradient Descent Search was completed and results collected. Both these searches made great improvements in the search time with only small or no perturbations in accuracy. These search methods make this optimal PID synthesis algorithm applicable to real time auto-tuning of PID controllers.

Knowledge of the plant model is necessary for these specific algorithms. In practice, this knowledge may not be available. There are promising results in [22] that a characterization of the stabilizing controller space can be achieved without knowledge of the plant model. However, it is not certain that optimal criteria like minimum H_2 and H_∞ metrics or any time domain metrics can be calculated from this method. At the very least, a “non-fragile” controller can be chosen based on the stabilizing space alone. This still provides a feasible path to the applicability of these algorithms to real time auto-tuning of PID controllers when knowledge of the plant is not available.

REFERENCES

[1] R. C. Dorf and R. H. Bishop, *Modern Control Systems.* Upper Saddle River, NJ: Prentice Hall, 12th ed., 2010.

[2] S. Bennett, *A History of Control Engineering 1800-1930.* London: Peter Peregrinus Ltd, 1979.

[3] S. Bennett, *A History of Control Engineering 1930-1955.* London: Peter Peregrinus Ltd, 1993.

[4] N. Minorsky, "Directional stability of automatically steered bodies," *Journal of the American Society for Naval Engineers*, vol. 34, no. 2, pp. 280–309, 1922.

[5] J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," *ASME Transactions - the American Society of Mechanical Engineers*, vol. 64, no. 8, pp. 759–768, 1942.

[6] K. Astrom and T. Hagglund, *PID Controllers: Theory, Design, and Tunning.* North Carolina: Instrument Society of America, 1995.

[7] B. Messner and D. Tilbury, "Control tutorials for MATLAB & SIMULINK." `http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID#1`. Accessed March, 2013.

[8] S. Bennett, *Autotuning of PID Controllers - A Relay Feedback Approach.* London: Springer-Verlag, 2nd ed., 2006.

[9] I. Nascu, R. De Keyser, S. Folea, and T. Buzdugan, "Development and evaluation of a pid auto-tuning controller," in *Automation, Quality and Testing, Robotics, 2006 IEEE International Conference on*, vol. 1, pp. 122–127, May.

[10] P. Wang and D. P. Kwok, "Auto-tuning of classical pid controllers using an advanced genetic algorithm," in *Industrial Electronics, Control, Instrumentation, and Automation, 1992. Power Electronics and Motion Control., Proceedings of the 1992 International Conference on*, pp. 1224–1229 vol.3, Nov.

[11] A. H. Jones and P. B. De Moura Oliveira, "Genetic auto-tuning of pid controllers," in *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*, pp. 141–145, Sep.

[12] C. N. Ko, T. L. Lee, H. T. Fan, and C. J. Wu, "Genetic auto-tuning and rule reduction of fuzzy pid controllers," in *Systems, Man and Cybernetics, 2006. SMC '06. IEEE International Conference on*, vol. 2, pp. 1096–1101, Oct.

[13] L. Fan and E. M. Joo, "Design for auto-tuning pid controller based on genetic algorithms," in *Industrial Electronics and Applications, 2009. ICIEA 2009. 4th IEEE Conference on*, pp. 1924–1928, May.

[14] Q. S. Lin, Y. F. Yao, and J. X. Wang, "Simulation and application of neural network pid auto-tuning controller in servo-system," in *Database Technology and Applications (DBTA), 2010 2nd International Workshop on*, pp. 1–4, Nov.

[15] S. Yanagawa and I. Miki, "Pid auto-tuning controller using a single neuron for dc servomotor," in *Industrial Electronics, 1992., Proceedings of the IEEE International Symposium on*, pp. 277–280, May.

[16] S. J. Lin, C. C. Tong, and N. K. Yang, "An auto-tuning grey-neuro-pid controller," in *Grey Systems and Intelligent Services, 2007. GSIS 2007. IEEE International Conference on*, pp. 845–850, Nov.

[17] S. S. Gade, S. B. Shendage, and M. D. Uplane, “On line auto tuning of pid controller using successive approximation method,” in *Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on*, pp. 277–280, March.

[18] M. Xu, S. Li, C. Qi, and W. Cai, “Auto-tuning of pid controller parameters with supervised receding horizon optimization,” *ISA Transactions*, vol. 44, no. 4, pp. 491–500, 2005, Oct.

[19] M. Zhuang and D. P. Atherton, “Automatic tuning of optimum pid controllers,” *Control Theory and Applications, IEE Proceedings D*, vol. 140, no. 3, pp. 216–224, May.

[20] Q. H. WU and B. W. HOGG, “On-line evaluation of auto-tuning optimal pid controller on micromachine system,” *International Journal of Control*, vol. 53, no. 4, pp. 751–769, 1991.

[21] L. H. Keel and S. P. Bhattacharyya, “A bode plot characterization of all stabilizing controllers,” *Automatic Control, IEEE Transactions on*, vol. 55, no. 11, pp. 2650–2654, Nov.

[22] L. H. Keel, B. Shafai, and S. P. Bhattacharyya, “A frequency response parametrization of all stabilizing controllers for continuous time systems,” in *American Control Conference, 2009. ACC ’09.*, pp. 3724–3729, June.

[23] B. Hori, J. Bier, and J. Eyre, “Use a microprocessor, a dsp, or both?,” in *Berkeley Design Technology, Embedded Systems Conference (ESC)*, April 2007.

[24] A. Datta, S. P. Bhattacharyya, and M. Ho, *Structure and Synthesis of PID Controllers*. London: Springer-Verlag, 2000.

[25] Texas Instruments (2008), “Overview of c++ support in TI compilers.” `http://processors.wiki.ti.com/index.php/Overview_of_C++_Support_in_TI_Compilers`, July 2008. Accessed Oct, 2012.

[26] D. Binner, “rpoly_ak1.cpp - program for calculating the roots of a polynomial of real coefficients.” `http://www.akiti.ca/rpoly\_ak1\_cpp.html`. Accessed Nov, 2012.

[27] M. A. Jenkins, “Algorithm 493: Zeros of a real polynomial [c2],” *ACM Trans. Math. Softw.*, vol. 1, pp. 178–189, June 1975.

[28] M. A. Jenkins and J. F. Traub, “A three-stage algorithm for real polynomials using quadratic iteration,” *SIAM J. Numer. Anal.*, vol. 7, pp. 545–566, Dec. 1970.

[29] P. Dawkins, “Paul’s online math notes - center of mass.” `http://tutorial.math.lamar.edu/Classes/CalcII/CenterOfMass.aspx`. Accessed Feb, 2013.

[30] W. J. Stevenson, *Operations Managemenet*. New York: McGraw-Hill Education, 11th ed., 2011.

[31] R. A. Johnson, *Modern Geometry: An Elementary Treatise on the Geometry of the Triangle and the Circle*. Boston, MA: Houghton Mifflin, 1929.

[32] M. Berkelaar, K. Eikland, and P. Notebaert, “lpsolve : Open source (mixed-integer) linear programming system.” `http://lpsolve.sourceforge.net/5.5/`. Accessed Oct, 2012.

[33] G. B. Dantzig and M. N. Thapa, *Linear Programming 2: Theory and Extensions*. New York: Springer-Verlag, 2003.

[34] A. H. Land and A. G. Doig, “An automatic method of solving discrete programming problems,” *Econometrica*, vol. 28, no. 3, pp. pp. 497–520, 1960.

[35] E. W. Weisstein, *CRC Concise Encyclopedia of Mathematics.* New York: Chapman and Hall/CRC, 2002.

[36] F. K. Avis, D., "A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra," *Discrete & Computational Geometry*, vol. 8, pp. 295–313, 1992.

[37] K. Zhou, J. C. Doyle, and K. Glover, *Robust and Optimal Control.* Upper Saddle River, NJ: Prentice Hall, 1995.

[38] A. Jameson, "Solution of the equation $AX + XB = C$ by inversion of an $M \times M$ or $N \times N$ matrix," *SIAM J. Appl. Math*, vol. 16, pp. 1020–1023, 1968.

[39] S. Bochkanov, "ALGLIB: a cross-platform numerical analysis and data processing library." `http://www.alglib.net/`. Accessed Nov, 2012.

[40] S. P. Bhattacharyya, H. Chapellat, and L. H. Keel, *Robust Control: The Parametric Approach.* Upper Saddle River, NJ: Prentice Hall, 1995.

[41] N. A. Bruinsma and M. Steinbuch, "A fast algorithm to compute the H_∞ of a transfer function matrix," *Systems & Control Letters*, vol. 14, no. 4, pp. 287 – 293, 1990.

[42] M. N. Belur and C. Praagman, "An efficient algorithm for computing the H_∞ norm," *Automatic Control, IEEE Transactions on*, vol. 56, pp. 1656 – 1660, July 2011.

[43] Texas Instruments (2010), "Tms320c66x dsp cpu and instruction set reference guide." `http://www.ti.com/lit/ug/sprugh7/sprugh7.pdf`. Accessed April, 2013.

[44] R. Bellman and K. L. Cooke, *Differential-Difference Equations.* New York/London: Academic Press Inc., 1963.

[45] L. S. Pontryagin, "On the zeros of transcendental functions with application," *American Mathematical Society Translation*, vol. 2, pp. 95–110, 1955.

[46] G. J. Silva, A. Datta, and S. P. Bhattacharyya, "Stabilization of first-order systems with time delay using the pid controller," in *American Control Conference, 2001. Proceedings of the 2001*, vol. 6, pp. 4650 –4655 vol.6, 2001.

[47] H. Vestermark, "cpoly.cpp - program for calculating the roots of a polynomial of complex coefficients." `http://www.hvks.com/Numerical/Downloads/cpoly_ver_1.cpp`. Accessed Dec, 2012.

APPENDIX A

SOURCE CODE FOR THE POLYNOMIAL CLASS

Listing A.1: Full Polynomial Class Declaration

```
#include "ComplexPolynomial.h"
class Polynomial {
        public:
        /* Constructors */
        Polynomial();
        Polynomial(int deg, double *ca);
        Polynomial(std::vector<double> cv);
        Polynomial(ComplexPolynomial cp);
        /* Copy Constructors */
        Polynomial(const Polynomial& p);
        Polynomial(int n);
        Polynomial(double d);

        double LIM_NORMPOLY(int npm);
        Polynomial POLY_NDERIV(int k);
        double POLY_NDERIV0_NORM(int k, int npm);
        double POLYVAL(double val);
        std::complex<double> POLYVAL(std::complex<double> val);
        std::vector<double> CPOLYVAL(double cval);
        std::vector< std::vector<double> > getRoots();

        /*Private Access Functions*/
        void setCoeff(double val, int idx);
        int getDegree();
        std::vector<double> getCoeffVec();
        double getLeadingCoeff();
        Polynomial removeLastCoeff();
        Polynomial removeLeadCoeff();
        Polynomial getEvenPart();
        Polynomial getOddPart ();
        Polynomial stojw();
        Polynomial sMinus();
        Polynomial POLYSHIFT(double a);
        Polynomial flipLR();
        std::vector<std::complex<double> > stoejphi(double phi);
        void printPoly();

        /*Operator Overloads*/
        Polynomial& operator=(const Polynomial &rhs);
        Polynomial& operator+=(const Polynomial &rhs);
        Polynomial& operator-=(const Polynomial &rhs);
        Polynomial& operator*=(const Polynomial &rhs);
        const Polynomial operator+(const Polynomial &other) const;
        const Polynomial operator-(const Polynomial &other) const;
        const Polynomial operator*(const Polynomial &other) const;
        bool operator==(const Polynomial &other) const;
        bool operator!=(const Polynomial &other) const;

        friend Polynomial operator*(double lhs, Polynomial &rhs);
        /* Destructor */
        ~Polynomial();
    private:
        std::vector<double> coeff;
        int degree;
};
```

Listing A.2: Full Polynomial Class Definition

```
/* C++ includes */
#include <iostream>
#include <limits>
#include <vector>
#include <complex>
#include <math.h>
/* User created C++ includes */
#include "Polynomial.h"
#include "polymath.h"
#include "rpoly_ak1.h"
/* Default Constructor
   Defines a Polynomial object with an empty coefficient
   vector and a degree of -1*/
Polynomial::Polynomial() {
    degree = -1;
}
/* Build a polynomial object from a given array of
   coefficients and the degree.*/
Polynomial::Polynomial(int deg, double *ca) {
    degree = deg;
    for (int i = 0; i <= deg; i++) {
        //Put coeficients into the vector coeff list
        coeff.push_back(ca[i]);
    }
}
/* Build a polynomial object from a given vector
   defining the coefficients of the polynomial */
Polynomial::Polynomial(std::vector<double> cv) {
    degree = cv.size() - 1;
    coeff = cv;
}
/* Copy Constructor */
Polynomial::Polynomial(const Polynomial& p) {
    degree = p.degree;
    coeff = p.coeff;
}
/*Conversion Constructors for implicit cast*/
/* int->Polynomial */
Polynomial::Polynomial(int n) {
    degree = 0;
    coeff.push_back((double)n);
}
/* double->Polynomial */
Polynomial::Polynomial(double d) {
    degree = 0;
    coeff.push_back(d);
```

```
}
/* ComplexPolynomial->Polynomial */
Polynomial::Polynomial(ComplexPolynomial cp) {
    std::vector<std::complex<double> > cmplxCoeff;
    degree = cp.getDegree();
    cmplxCoeff = cp.getCoeffVector();
    for (int i = 0; i < cmplxCoeff.size(); ++i)
        coeff.push_back(cmplxCoeff[i].real());
}
/* This function is a wrapper for the code found here:
       http://www.akiti.ca/rpoly_ak1_cpp.html
    This code is free for public use and is based on the Jenkins-Traub algorithm */
std::vector< std::vector<double> > Polynomial::getRoots() {
    int deg = this->degree; // The degree of the polynomial to be solved
    double zeroi[MAXDEGREE], zeror[MAXDEGREE], op[MDP1]; // Coefficient vectors
    int i; // vector index
    /* Input the polynomial coefficients from the
       Polynomial Object and put them in the op vector */
    for (i = 0; i <= deg; i++)
        op[i] = this->coeff[i];
    rpoly_ak1(op, &deg, zeror, zeroi);
    std::vector< std::vector<double> > roots(deg, std::vector<double>(2));
    for (i = 0; i < deg; i++) {
            roots[i][0] = zeror[i];
            roots[i][1] = zeroi[i];
    }
    return roots;
```

```
}
/* PID Algorithm Specific Functions */
/* LIM_NORMPOLY
   Gives the result of the limit of the normalized form of a
   polynomial at INF. The normalized polynomial is defined as
                    p(w)/(1+w^2)^(mpn/2) */
double Polynomial::LIM_NORMPOLY(int npm) {
    if (this->degree < npm){
        return 0;
    }
    else if (this->degree > npm) {
        return SGN((this->coeff).front())*INF;
    }
    else {//if (this->degree == npm) {
        return (this->coeff).front();
    }
}
/* POLY_NDERIV
   return the kth derivative of a polynomial */
Polynomial Polynomial::POLY_NDERIV(int k) {
    Polynomial result; //Polynomial Object to hold the result
    //Copy values of the calling polynomial into result
    result.degree = this->degree;
    result.coeff = this->coeff;
    //Compute the derivative
    for (int i = 0; i < k; i++) { //derivative loop
        for(int j = 0; j < result.degree; j++) { //term modifer loop
```

```
            result.coeff[j] *= (result.degree - j);
        }
        result.coeff.pop_back();
        result.degree -= 1;
    }
    return result;
}
/* POLY_NDERIV0_NORM
   Gives the result of the kth derivative of the normalized form of a
   polynomial evaluated at 0. the normalized polynomial is defined as
                    p(w)/(1+w^2)^(npm/2) */
double Polynomial::POLY_NDERIV0_NORM(int k, int npm) {
    std::vector<double> ct, term;
    double result = 0.0;
    if (k == 1) {
        //easy case handle it quickly
        return this->coeff[this->degree];
    }
    else {
        for(int i = k; i >= 0; i -= 2)
            ct.push_back(this->coeff[this->degree - i]);
        term.push_back(FACTORIAL(k)); //first term
        double a = 1.0;
        for (int i = 1; i <= (k - floor((k-1)/2.0) - 1); i++) {
            a *= ( npm/2.0 + (i-1) );
            term.push_back( a * (pow(-1.0, i)*FACTORIAL(k)) / FACTORIAL(i) );
        }
```

```
            for (int i = 0; i < (int)term.size(); i++)
                result += term[i]*ct[i];
        }
        return result;
}
/* POLYVAL
   Evaluates the Polynomial given a real value */
double Polynomial::POLYVAL(double val) {
        double result = 0.0;
        for(int j = 0; j <= this->degree; j++)
            result += this->coeff[j]*pow(val,this->degree - j);
        return result;
}
/* POLYVAL
   Evaluates the Polynomial given a complex value */
std::complex<double> Polynomial::POLYVAL(std::complex<double> val) {
        std::complex<double> result;
        for(int j = 0; j <= this->degree; j++)
            result += this->coeff[j]*pow(val, this->degree - j);
        return result;
}
/* CPOLYVAL(double)
   Evaluates a polynomial given a complex parameter
   with the real part = 0. This is a specialized routine
   to handle a special case within the PID algorithm */
std::vector<double> Polynomial::CPOLYVAL(double cval) {
        std::vector<double> result(2, 0.0);
```

```
    for(int j = 0; j < this->degree; j++) {
        if ((this->degree - j)%2 == 0 )
            result[0] += this->coeff[j]*pow(cval, this->degree - j)* ...
                pow(-1.0, (this->degree - j)/2);
        else
            result[1] += this->coeff[j]*pow(cval, this->degree - j)* ...
                pow(-1.0, (this->degree - j - 1)/2);
    }
    result[0] += this->coeff[this->degree];
    return result;
}
/* getEvenPart()
   Returns the even part of the Polynomial */
Polynomial Polynomial::getEvenPart() {
    Polynomial p;
    p.degree = this->degree;
    p.coeff = this->coeff;
    int q = p.degree % 2;
    if (q == 1) {
        p.coeff.erase(p.coeff.begin());
        p.degree -= 1;
    }
    std::vector<double>::iterator it;
    for(it = p.coeff.begin()+1; it < p.coeff.end(); it+=2)
        *it = 0;
    return p;
}
```

```

/* getOddPart()
   Returns the odd part of the Polynomial */
Polynomial Polynomial::getOddPart() {
    Polynomial p;
    p.degree = this->degree;
    p.coeff = this->coeff;
    int q = p.degree % 2;
    p.removeLastCoeff();
    if (q == 0 && p.degree > -1) {
        p.coeff.erase(p.coeff.begin());
        p.degree -= 1;
    }
    if (p.degree == -1) {
        p.coeff.push_back(0.0);
        p.degree = 0;
    }
    std::vector<double>::iterator it;
    for(it = p.coeff.begin()+1; it < p.coeff.end(); it+=2)
        *it = 0;
    return p;
}

/* stojw()
   converts the s parameter to jw and completes proper
   exponent math for signs */
Polynomial Polynomial::stojw() {
    for(int i = 0; i < this->degree; i+=2)
        this->coeff[i] = this->coeff[i]* ...
            (((int)floor( (this->degree - i)/2.0 ) % 2) * -2 + 1);
    return *this;
}

/* stoejphi
   converts the s parameter to e^jp and completes proper
   exponent math for signs */
std::vector<std::complex<double> > Polynomial::stoejphi(double phi) {
    std::complex<double> cc(cos(phi), sin(phi));
    std::vector<std::complex<double> > Ps1ej(this->degree + 1);
    for (int k = this->degree; k >= 0; --k)
        Ps1ej[this->degree - k] = pow(cc, k)*this->coeff[this->degree - k];
    return Ps1ej;
}

/* sMinus()
   converts the s parameter to -s and completes proper
   exponent math for signs */
Polynomial Polynomial::sMinus() {
    Polynomial p;
    p.degree = this->degree;
    p.coeff = this->coeff;
    for(int i = 0; i < p.coeff.size(); ++i)
        p.coeff[i] *= pow(-1.0,(p.degree-i));
    return p;
```

```
}
/* POLYSHIFT
   Given a polynomial p(x), returns the polynomial p(x−a) */
Polynomial Polynomial::POLYSHIFT(double a) {
    double tarray[] = {1, −a};
    Polynomial T(1, tarray);
    Polynomial TMP;
    int p = this−>degree;
    Polynomial P(0); //constant polynomial equal to 0
    for (int i = 0; i < this−>degree − 1; ++i) {
        TMP = T;
        for (int j = 1; j < p; ++j)
            TMP = TMP*T;
        TMP = this−>coeff[i]*TMP;
        P = P + TMP;
        p = p − 1;
    }
    //deal with x term and constant term
    T = this−>coeff[this−>degree−1]*T + this−>coeff.back();
    P = P + T;
    return P;
}
/* flipLR()
   Flip the polynomials coefficient vector */
Polynomial Polynomial::flipLR() {
    Polynomial F;
    for (int i = this−>coeff.size()−1; i >= 0; −−i)
```

```
            F.coeff.push_back(this->coeff[i]);
    F.degree = this->degree;
    return F;
}
/*Private Access Functions*/
void Polynomial::setCoeff(double val, int idx) {
    this->coeff[idx] = val;
}
int Polynomial::getDegree() {
    return this->degree;
}
std::vector<double> Polynomial::getCoeffVec() {
    return this->coeff;
}
double Polynomial::getLeadingCoeff() {
    return (this->coeff).front();
}
Polynomial Polynomial::removeLastCoeff() {
    this->coeff.pop_back();
    this->degree -= 1;
    return *this;
}
Polynomial Polynomial::removeLeadCoeff() {
    this->coeff.erase(this->coeff.begin());
    this->degree -= 1;
    return *this;
}
```

```
void Polynomial::printPoly() {
    int i;
    for(i = 0; i < this->degree; i++)
        std::cout << this->coeff[i] << "x^" << this->degree - i << " + ";
    std::cout << this->coeff[i] << "\n";
}
/* Operator Overload Implementation */
Polynomial& Polynomial::operator=(const Polynomial &rhs) {
    // Only do assignment if RHS is a different object from this.
    if (this != &rhs) {
        degree = rhs.degree;
        coeff = rhs.coeff;
    }
    return *this;
}
Polynomial& Polynomial::operator+=(const Polynomial &rhs) {
    // Do the compound assignment work.
    // adds two polynominals of any length
    std::vector<double> rhs_coeff;
    rhs_coeff = rhs.coeff;

    if (this->degree > rhs.degree) {
        unsigned int diff = this->degree - rhs.degree;
        //Make the RHS equal in degree by adding 0's
        rhs_coeff.insert(rhs_coeff.begin(), diff, 0);
    }
    else if (this->degree < rhs.degree) {
        unsigned int diff = rhs.degree - this->degree;
        //Make the LHS equal in degree by adding 0's
        this->coeff.insert(this->coeff.begin(), diff, 0);
        this->degree = rhs.degree;
    }

    for(int i = 0; i <= this->degree; i++)
        this->coeff[i] += rhs_coeff[i];

    return *this;
}
Polynomial& Polynomial::operator-=(const Polynomial &rhs) {
    // Do the compound assignment work.
    // adds two polynominals of any length
    std::vector<double> rhs_coeff;
    rhs_coeff = rhs.coeff;

    if (this->degree > rhs.degree) {
        unsigned int diff = this->degree - rhs.degree;
        //Make the RHS equal in degree by adding 0's
        rhs_coeff.insert(rhs_coeff.begin(), diff, 0);
    }
    else if (this->degree < rhs.degree) {
        unsigned int diff = rhs.degree - this->degree;
        //Make the LHS equal in degree by adding 0's
        this->coeff.insert(this->coeff.begin(), diff, 0);
        this->degree = rhs.degree;
    }
    for(int i = 0; i <= this->degree; i++)
        this->coeff[i] -= rhs_coeff[i];

    return *this;
}
Polynomial& Polynomial::operator*=(const Polynomial &rhs) {
    // Implements convolution of the coefficent vectors
    int kmin;
    int kmax;
    int n = this->coeff.size() + rhs.coeff.size() - 1; // length of result
    std::vector<double> tmp_lhs(n, 0.0);

    for (int i = 0; i < n; i++) {
        if (i >= rhs.degree)
            kmin = i - (rhs.degree);
        else
            kmin = 0;

        if (i < this->degree)
            kmax = i;
        else
            kmax = this->degree;

        for (int k = kmin; k <= kmax; k++)
            tmp_lhs[i] += (this->coeff[k])*(rhs.coeff[i-k]);
    }
    this->coeff = tmp_lhs;
    this->degree = tmp_lhs.size() - 1;
    return *this;
}
const Polynomial Polynomial::operator+(const Polynomial &other) const {
    return Polynomial(*this) += other;
}
const Polynomial Polynomial::operator-(const Polynomial &other) const {
    return Polynomial(*this) -= other;
}
const Polynomial Polynomial::operator*(const Polynomial &other) const {
    return Polynomial(*this) *= other;
}
bool Polynomial::operator==(const Polynomial &other) const {
    double tmp = 0;
    if (this->degree == other.degree) {
        for(int i = 0; i <= this->degree; i++)
            tmp += (this->coeff[i] - other.coeff[i]);
        if (tmp == 0)
            return true;
        else
            return false;
    }
    else {
        return false;
    }
}
```

```
bool Polynomial::operator!=(const Polynomial &other) const {
    return !(*this == other);
}
/* Destructor */
Polynomial::~Polynomial() { /*empty destructor*/}

/* Free Operator overload outside of the class */
/* Handles multiplication by a constant on the LHS */
Polynomial operator*(double lhs, Polynomial &rhs){
    Polynomial *tmp = new Polynomial();
    tmp->degree = rhs.degree;
    for (int i = 0; i < rhs.coeff.size(); i++)
        tmp->coeff.push_back(lhs * rhs.coeff[i]);
    return *tmp;
}
```