Centro de Investigación Científica y de Educación Superior de Ensenada, Baja California

Maestría en Ciencias en Electrónica y Telecomunicaciones con orientación en Instrumentación y Control

---

Algoritmos bioinspirados para la sintonización de sistemas de control con retardo

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de Maestro en Ciencias

Presenta:

René Míchel Martínez Trujillo

Ensenada, Baja California, México

2018

Tesis defendida por

René Míchel Martínez Trujillo

y aprobada por el siguiente Comité

Dr. Luis Alejandro Márquez Martínez
Codirector de tesis

Dr. Carlos Alberto Brizuela Rodríguez
Codirector de tesis

Dra. María del Carmen Maya Sánchez

Dra. Meritxell Riquelme Pérez

Dr. Daniel Sauceda Carvajal
Coordinador del Posgrado en Electrónica y Telecomunicaciones

Dra. Rufina Hernández Martínez
Directora de Estudios de Posgrado

Resumen de la tesis que presenta René Míchel Martínez Trujillo como requisito parcial para la obtención del grado de Maestro en Ciencias en Electrónica y Telecomunicaciones con orientación en Instrumentación y Control.

Algoritmos bioinspirados para la sintonización de sistemas de control con retardo

Resumen aprobado por:

Dr. Luis Alejandro Márquez Martínez
Codirector de tesis

Dr. Carlos Alberto Brizuela Rodríguez
Codirector de tesis

El controlador Proporcional-Integral-Derivativo (PID) es el esquema de control más dominante en la industria. Sin embargo, su sintonización óptima no es evidente y mucho menos cuando el sistema presenta retardos en el tiempo. Los retardos temporales aparecen con frecuencia en los sistemas reales. Estos retardos deben considerarse en el modelo al momento de sintonizar el controlador, ya que no hacerlo puede degradar el desempeño del sistema de control. Actualmente existe una gran variedad de técnicas para sintonizar el controlador PID, pero no todas toman en cuenta a los retardos, lo que conlleva a una mala sintonización y por ende un mal desempeño. El problema de sintonización de controladores se ha modelado como un problema de optimización que en la mayoría de los casos no cumple con restricciones como convexidad y diferenciabilidad, y hace imposible la aplicación de métodos exactos de optimización. Por ello, en los últimos años se ha abordado el problema con técnicas metaheurísticas como las basadas en el cómputo bioinspirado, ya que han mostrado buenos resultados al resolver problemas complejos de optimización. En este trabajo de tesis se presenta una metodología para sintonizar automáticamente los parámetros de un controlador PID aplicados a sistemas con retardo, con el objetivo de lograr un desempeño cercano al óptimo. El problema de la sintonización se aborda como un problema de optimización tanto mono como multi-objetivo, utilizando y comparando dos metaheurísticas: evolución diferencial y optimización por enjambre de partículas. Como funciones objetivo, se usaron criterios en el dominio del tiempo como lo son los índices de desempeño basados en la integral de funciones del error, el sobreimpulso y el tiempo de asentamiento. Este procedimiento se aplicó a un sistema lineal de segundo orden con retardo en la entrada. Los resultados de simulación numéricos permitieron identificar al criterio de optimización con el que se obtiene el mejor desempeño (bajo tres criterios particulares) del sistema de control. Se confirmó también el grado de conflicto existente entre el sobreimpulso, tiempo de asentamiento y esfuerzo de control. Además los resultados obtenidos en cuanto a sobreimpulso, tiempo de asentamiento y esfuerzo de control fueron mejores que los logrados con un esquema comercial de una empresa líder en el ámbito académico e industrial.

Palabras clave: retardo temporal, sintonización, controlador PID, algoritmos metaheurísticos, optimización

Abstract of the thesis presented by René Míchel Martínez Trujillo as a partial requirement to obtain the Master of Science degree in Electronics and Telecommunications with orientation in Intrumentation and Control.

Bio-inspired algorithms for the tuning of time-delay control systems

Abstract approved by:

Dr. Luis Alejandro Márquez Martínez	Dr. Carlos Alberto Brizuela Rodríguez
Thesis Co-Director	Thesis Co-Director

The Proportional-Integral-Derivative controller is the most dominant control scheme in the industry. However, its optimal tuning is not evident, in particular when the system has time delays. Temporary delays frequently appear in real systems. These delays must be considered in the model when tuning the controller since not doing so can degrade the performance of the control system. Currently, there is a wide variety of techniques to tune the PID controller, however not all of them take delays into account, leading to poor tuning and therefore poor performance. The controller tuning problem has been modeled as an optimization problem that in most cases does not comply with constraints such as convexity and differentiability, and makes it impossible the application of exact optimization methods. Therefore, in recent years the problem has been addressed with metaheuristic techniques such as bioinspired computation methods, since they have shown good results when solving complex optimization problems. In this thesis, a methodology is presented to automatically tune the parameters of a PID controller applied to systems with time delay, with the aim of achieving close to the optimum solutions. The problem of tuning is addressed as a problem of both mono and multi-objective optimization, using and comparing two metaheuristics: differential evolution and particle swarm optimization. As objective functions, criteria in the time domain are used, as are performance indices based on the integral of error functions, the overshoot and the settling time. This procedure is applied to a second-order linear system with input time delay. The numerical simulation results allowed to identify the optimization criterion with which the best performance (under three particular criteria) of the control system obtained. The degree of conflict between the overshoot, settling time and control effort was also confirmed. Furthermore, the results obtained concerning overshoot, settling time, and control effort were better than those achieved by a commercial scheme of a leading company in the academic and industrial sector.

Keywords: time delay, tuning, PID controller, metaheuristic algorithms, optimization

Dedicatoria

A mi familia por todo el apoyo incondicional y por creer en mi, en especial a mi Madre, porque por ella soy lo que soy.

Agradecimientos

Al Centro de Investigación Científica y de Educación Superior de Ensenada por aceptarme en unos de sus prosgrados y permitirme estudiar una maestría en electrónica y telecomunicaciones.

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por brindarme el apoyo económico para realizar mis estudios de maestría. No. de becario: 613823.

A toda mi familia por el apoyo incondicional a lo largo de este camino. En especial a mi madre Esther, quien con su ejemplo de vida y apoyo incondicional, a pesar de todas las dificultades, siempre brindó su amor, su tiempo y su ayuda, para que mis hermanos y yo nos superaramos día con día. A mis abuelas Esther y María, por todos sus consejos y palabras de aliento. A mi tío Ricardo, por sus palabras motivadoras y su apoyo. A Teodoro Maldonado por todo su apoyo. A mi Padre René, por enseñarme que a veces la vida no es justa, pero siempre hay que salir adelante, pese a todo. Un reconocimiento especial a mis abuelos Santiago† y Santos†, por creer en mi y motivarme a seguir adelante. Con todos ustedes estaré eternamente agradecido.

A mi querida novia, Marisela, por siempre permanecer a mi lado en esta etapa y desde hace nueve años en mi vida, dando todo su amor, cariño, y apoyo. Sin ti no sé que hubiera sido de mí en mis momentos de soledad, de locura y de pobreza. Con este, es otro logro más que compartimos. Te estaré eternamente agradecido, te amo.

“Soy prisionero del beso que nunca debiste darme" *- A.S. -*

A mis directores de tesis, Dr. Alejandro Márquez y Dr. Carlos Brizuela, por ustedes sé que aún existen excelente docentes que comparten su pasión. Gracias por su gran paciencia, su comprensión y sobre todo por su apoyo.

A mi comité de tesis, Dra. Carmen Maya y Dra. Meritxell, por sus aportes en el

desarrollo del trabajo y sobre todo en el ámbito profesional y personal.

A todos mis profesores durante el posgrado, Dr. Miguel Alonso, Dr. David Covarrubias, Dr. Miguel Murillo, Dr. Adrián Arellano, Dr. Ricardo Cuesta, Dra. Carmen Maya, M en C. Moisés Castro y Dr. César Cruz. Un agradecimiento especial para los profesores, Dr. Joaquín Álvarez, Dr. Jonatan Peña y Dr. Yury Orlov. Gracias a todos por su conocimiento y consejos.

A todo mis compañeros y amigos de generación, Jessica, Edén, Adán, Gustavo, Fernando, Ernesto “*El Pariente*", Edwin, Gerardo, Eduardo y Víctor, por su compañia, ayuda y amistad durante estos años.

A mis compañeros del cubículo 244, Abimael y Gerardo, por sus consejos y apoyo durante la maestría.

A todos mis compañeros y amigos de CICESE, en especial a Mízquez y Rolando, por su amistad, las experiencias vividas y el haber compartido como rivales la cancha de fútbol. Una mención especial a Lourdes “Lulú Aguillón y a Karla por ayudarme en momentos complicados.

A todos los Zenin, Carolina, Lizeth, David, Edgar, Torres, Estrella, Camilo, Abrahm, Adriana, Cova y Bojo, por todos los torneos que compartimos las canchas de fútbol y por pasar grandes momentos alegres cómo aquel torneo de la UNAM que ganamos.

A la manada, Marisela y Sofía, por tan bellos momazos que alegraban mis días.

A la Sra. Rosa Lara y al Sr. Samuel López, por todos los consejos, “ridesz apoyo que me brindaron durante tantos años.

Al Dr. Luis Ahedo, por motivarme a seguir en el camino de la ciencia.

A mis amigos del alma, Carmen “*Pagua*", Rodolfo “*Rodo*", Miguel “*Mike*", *Orlando*, Juan Carlos “*JC*", Benjamín “*Benja*", por su amistad incondicional, por todos los buenos

momentos que hemos pasado desde la universidad, por las aventuras y todas las tardes de banqueteras que pasamos juntos, les agradezco de todo corazón su amistad, les prometo que siempre estaré con y para ustedes.

A los médicos, Luz Ochoa y Jose Rubio, y a la psicóloga Yahaira Castañeda, por ayudarme a sobrellevar mis problemas de salud y darme la tranquilidad para culminar con mi trabajo.

“Que la fuerza los acompañe" - *Star Wars* -

Tabla de contenido

Página

Resumen en español ii
Resumen en inglés iii
Dedicatoria iv
Agradecimientos v
Lista de figuras x
Lista de tablas xii
Lista de Abreviaciones xiii

Capítulo 1. Introducción
1.1. Antecedentes 2
1.2. Objetivos 4
1.2.1. Objetivo general 4
1.2.2. Objetivos específicos 4
1.3. Propuesta de solución 4

Capítulo 2. Preliminares
2.1. Sistemas con retardo 6
2.1.1. Modelo del sistema considerado 7
2.2. Estrategias de optimización 8
2.2.1. Optimización mono-objetivo 8
2.2.2. Optimización multi-objetivo 9
2.3. Metaheurísticas de optimización 10
2.3.1. Evolución diferencial 10
2.3.2. Optimización por enjambre de partículas 14
2.3.3. Evolución diferencial para la optimización multi-objetivo 16
2.4. Criterios de optimización 18
2.4.1. Criterios en el dominio de la frecuencia 18
2.4.2. Criterios en el dominio del tiempo 19
2.4.2.1. Índices de desempeño 20
2.4.2.2. Parámetros de la forma de la respuesta 22
2.5. Controlador PID 22
2.6. Sintonización Ziegler-Nichols 25
2.6.1. Primer método de Ziegler-Nichols 25
2.6.2. Segundo método de Ziegler-Nichols 26

Capítulo 3. Metodología propuesta
3.1. Simulación 27
3.1.1. Parámetros iniciales 27
3.1.2. Población inicial 29
3.1.3. Evaluar población 29
3.1.4. Evaluar función objetivo 30

Tabla de contenido (continuación)

3.1.5. Paro 31
3.1.6. Parámetros del PID 31
3.1.7. Truncar y obtener frentes no dominados 32

Capítulo 4. Experimentos y Resultados
4.1. Especificaciones de hardware y software 33
4.1.1. Parámetros de los algoritmos 34
4.2. Optimización mono-objetivo 35
4.2.1. Evolución diferencial 36
4.2.2. Optimización por enjambre de partículas 41
4.3. Comparativa de desempeño de los algoritmos 46
4.4. Optimización multi-objetivo 49
4.4.1. Optimización con dos funciones objetivo 49
4.4.2. Optimización con tres funciones objetivo 52

Capítulo 5. Conclusiones
5.1. Trabajo a futuro 55

Literatura citada 56

Anexos 61

Anexo A 61
A.1. Programas en Python 61
A.1.1. Metaheurísticas de optimización 61
A.1.1.1. Evolución Diferencial 61
A.1.1.2. Optimización por enjambre de partículas 64
A.1.1.3. Evolución diferencial para la optimizacin multi-objetivo (2 criterios) 67
A.1.1.4. Evolución diferencial para la optimizacin multi-objetivo (3 criterios) 69
A.1.2. Comunicación entre Python y C++ 69
A.1.3. Obtención de código en lenguaje C++ 70

Anexo B 73
B.1. Programa en C++ y modificaciones al código 73

Lista de figuras

Figura Página

1. Función de costo bidimensional. 12
2. Proceso de cruzamiento. 13
3. Ejemplo tridimensional del movimiento de las partículas: proceso con cero iteraciones (izquierda), después de N iteraciones (derecha). 16
4. Ejemplo bidimiensional del movimiento de la partícula en el espacio de búsqueda. 16
5. Características de una respuesta al escalón. 23
6. Esquema básico de implementación del controlador PID. 23
7. Caracterización de la respuesta en escalón en el método de la respuesta a un escalón. 25
8. Diagrama de flujo de la sintonización del controlador PID con enfoque mono-objetivo (izquierda) y multi-objetivo (derecha). 28
9. Diagrama de bloques de la sintonización de un controlador PID 28
10. Diagramas de flujo del bloque para evaluar función objetivo de cada algoritmo. 31
11. Respuesta del sistema en lazo abierto de segundo orden a una entrada escalón. 36
12. Evolución de las funciones de costo optimizadas con evolución diferencial. 38
13. Respuestas del sistema de control a una entrada escalón optimizadas con evolución diferencial para los criterios de ITAE, ISE, ISTAE, ISTSE, M_p, t_s incluyendo los métodos KPSO, `pidtune()` y Z-N. 39
14. Respuestas del sistema de control a una entrada escalón optimizadas con evolución diferencial para los criterios de IAE, IE, ITSE y t_s. 40
15. Señales del control PID sintonizado con evolución diferencial utilizando los criterios de optimización de IAE, IE, ISE e ITSE, incluyendo los métodos de `pidtune` y Z-N. 40
16. Señales del control PID sintonizado con evolución diferencial utilizando los criterios de optimización de ISTAE, ISTSE, M_p y t_s, incluyendo el método KPSO. 41
17. Evolución de las funciones de costo optimizadas con optimización por enjambre de partículas. 43
18. Respuestas del sistema de control a una entrada escalón optimizada con optimización por enjambre de partículas para los criterios de ITAE, ISTAE, ISTSE y M_p, incluyendo los métodos KPSO, `pidtune()` y Z-N. 44
19. Respuestas del sistema de control a una entrada escalón optimizada con optimización por enjambre de partículas para los criterios de IAE, ITSE, ISE y t_s. 44

Lista de figuras (continuación)

Figura Página

20. Señales del control PID sintonizado con optimización por enjambre de partículas utilizando los criterios de optimización de ITAE, IAE, IE, ISE e ITSE, incluyendo los métodos de `pidtune` y Z-N. 45

21. Señales del control PID sintonizado con optimización por enjambre de partículas utilizando los criterios de optimización de ISTAE, ISTSE, M_p y t_s, incluyendo el método KPSO. 45

22. Comparación de mejor resultado entre DE y PSO. 46

23. Acercamiento de comparación de mejor resultado entre DE y PSO. 47

24. Comparación de resultado promedio entre DE y PSO. 48

25. Acercamiento de comparación de resultado promedio entre DE y PSO. . . 48

26. Frente no dominado de soluciones óptimas obtenido con la metaheurística DEMO. 51

27. Frente no dominado de soluciones óptimas obtenido con la metaheurística DEMO. 53

Lista de tablas

Tabla Página

1. Parámetros del controlador para el método de la respuesta a un escalón. 26
2. Parámetros del controlador para el método de la respuesta en frecuencia. 26
3. Parámetros del controlador PID y características principales. 36
4. Parámetros del controlador PID y características principales de la respuesta del sistema con retardo obtenidos con el algoritmo evolución diferencial. 37
5. Estadísticas de la optimización de las funciones de costo con DE. . . 38
6. Estadísticas del tiempo de asentamiento de las respuestas al escalón con DE. 38
7. Parámetros del controlador PID y características principales de la respuesta del sistema con retardo obtenidos con el algoritmo optimización por enjambre de partículas. 42
8. Estadísticas de la optimización de las funciones de costo con PSO. . 42
9. Estadísticas del sobreimpulso y tiempo de asentamiento de las respuestas al escalón con PSO. 43
10. Tabla con los parámetros del controlador PID y características principales de la respuesta del sistema optimizando con dos funciones objetivo. 51

Lista de abreviaciones y símbolos

Abreviaciones, por sus siglas en inglés.

DE	Evolución diferencial.
DEMO	Evolución diferencial para la optimización multiobjetivo.
IAE	Integral del valor absoluto del error.
IE	Integral del error.
IMC	Control por modelo interno.
ISE	Integral del error cudrático.
ISTAE	Integral del tiempo al cuadrado multiplicado por el error absoluto.
ISTSE	Integral del tiempo al cuadrado multiplicado por el error cuadrático.
ITAE	Integral del tiempo multiplicado por el valor absoluto del error.
ITSE	Integral del tiempo multplicado por el error cuadrático.
LQR	Regulador lineal cuadrático.
PID	Proporcional-integral-derivativo.
PSO	Optimización por enjambre de partículas.
Z-N	Ziegler-Nichols.

Símbolos.

K_p	Acción proporcional
M_p	Sobreimpulso.
T_d	Acción derivativa.
T_e	Tiempo de ejecución.
T_i	Acción integral.
t_s	Tiempo de asentamiento.
$\|\|\cdot\|\|_1$	Norma uno.
$\|\|\cdot\|\|_2$	Norma dos.
$\|\|\cdot\|\|_\infty$	Norma infinito.

Capítulo 1 Introducción

En sistemas reales, como los biológicos, químicos o mecánicos, es común encontrar retardos de forma intrínseca o inclusive que estos mismos se induzcan debido al transporte de energía o de información, así como a los tiempos de reacción de actuadores y sensores. Estos retardos pueden afectar al estado del proceso, a la entrada o a la salida (ver sección 2.1). Dichos retardos si son ignorados pueden degradar el desempeño de los controladores e incluso provocar inestabilidad en el sistema de control, ya que generalmente presentan cierta sensibilidad a este fenómeno. Sin embargo, si estos son incluidos en los modelos matemáticos, conducen a ecuaciones diferenciales y de diferencias, las cuales son más complicadas de abordar.

Además de estos sistemas, se han propuesto controladores que introducen voluntariamente retardos en un sistema (Michiels y Niculescu, 2007).Esto, permite aprovechar la mayor riqueza dinámica que se obtiene o tomar en cuenta el tiempo de retardo que se induce, por ejemplo, al filtrar señales. Un ejemplo de esto son los controladores tipo proporcional-integral-derivativo (PID) donde se considera un término de velocidad retardada (Skogestad, 2003). Todo esto ha conducido al desarrollo de nuevos esquemas de control cuya sintonización no es evidente y la cual se realiza por el especialista a partir de prueba y error.

La búsqueda, por este método, de los valores óptimos de los parámetros a sintonizar resulta limitada debido a que es posible explorar solo una porción pequeña del espacio posible de soluciones. Al modelar el problema de sintonización como uno de optimización resulta en un modelo que no cumple restricciones de diferenciabilidad, convexidad y en muchos casos ni siquiera continuidad, ya sea por controladores discontinuos o conmutados, o por el uso de criterios particulares, lo cual hace imposible la aplicación de métodos exactos de optimización. Esto motiva a abordar el problema de sintonización automática de parámetros mediante el uso de técnicas metaheurísticas como las basadas en cómputo bioinspirado (ver sección 2.3) que han mostrado su viabilidad al resolver problemas complejos de optimización donde se requieren optimizar uno o varios objetivos al mismo tiempo. Contar con un esquema para la sintonización de parámetros permitirá además, al mismo tiempo, evaluar la calidad de las nuevas técnicas de control propuestas para esta clase de sistemas.

1.1. Antecedentes

Hasta las primeras décadas del siglo XX, en la mayoría de los procesos los controladores utilizados eran puramente una acción proporcional; aunque ya se había desarrollado y existían mecanismos que la implementaban, la acción integral, no era tan utilizada. Fue hasta en los inicios de los años treinta que se agregó esta acción de control y a finales de esta década se adicionó la acción derivativa, formando el controlador PID (Bennett, 2001). El controlador PID como lo conocemos hoy en día se había analizado y discutido anteriormente por Minorsky (1922). A pesar de ser un controlador primitivo,en la actualidad es utilizado en más del 90% de aplicaciones industriales (Åström y Hägglund, 2001), debido a su construcción simple y otras características como robustez y facilidad de implementación.

Para sistemas con retardo, se necesitan controladores que sean capaces de asegurar estabilidad y que las respuestas cumplan con los requisitos de diseño, como sobreimpulso, tiempo de asentamiento y robustez, entre otros. Sin embargo, el problema en el diseño del controlador surge al querer aplicar directamente un ánalisis de control clásico como se haría con los sistemas sin retardo. Por ello, a través de los años se han desarrollado varios métodos, tanto analíticos como gráficos, y reglas de ajuste (prueba y error), para determinar los parámetros del controlador PID.

En la literatura existen más de 1,500 fórmulas de sintonización (O'Dwyer, 2009). Un referente clásico es la llamada heurística de Ziegler-Nichols (Z-N) (Ziegler y Nichols, 1942), la cual surgió durante los avances tecnológicos de la Segunda Guerra Mundial. Esta consta de dos métodos analíticos para determinar los parámetros del controlador. Estos se obtienen por medio de dos variables que caracterizan la dinámica del sistema en respuesta a una entrada escalón y en respuesta a frecuencia. Es un método sencillo y eficaz para obtener los valores de los parámetros con cálculos rápidos.

Una de las variantes surgida a raíz de esta heurística, es la de Chien *et al.* (1952), donde propone diferentes reglas de sintonía para perturbaciones en la carga y respuesta a cambios en el punto de referencia, otorgando mayor robustez en lazo cerrado. Otra propuesta es la de Cohen y Coon (1953), en la cual su principal criterio es el rechazo de las perturbaciones de carga, asignando tres polos en lazo cerrado, uno de ellos real y los otros dos polos complejos. A estas heurísticas se les conoce

como reglas de ajuste, o a prueba y error, de una manera informal.

Como la heurística de Z-N presenta poca robustez, varios trabajos abordaron dicha problemática y la resolvieron. En algunos de ellos se utilizan las mismas ideas que en Z-N, pero combinadas con el uso de restricciones de robustez (Hägglund y Åström, 2002; Åström y Hägglund, 2004). Otro método desarrollado es el de la ubicación de polos, que se enfoca en posicionar los polos de lazo cerrado en locaciones específicas (Persson, 1992; Åström y Hägglund, 1995).

También existen otros métodos de diseño, conocidos como iterativos. Algunos de ellos están basados en el margen de ganancia y margen de fase, y se llevan a cabo por medio de un análisis de forma gráfica (Srivastava y Pandit, 2016a), o de forma analítica (Ho *et al.*, 1995; Hu *et al.*, 2011). Otros están basados en el principio del control con modelo interno (IMC, por sus siglas en inglés) (Rivera *et al.*, 1986; Wang *et al.*, 2016), ya que siguiendo su procedimiento de diseño y haciendo hipótesis especiales se obtiene un controlador PID. La robustez se considera explícitamente en el diseño, por lo cual, su principal ventaja es la robustez contra perturbaciones e inexactitudes del modelo de la planta.

Existen otros métodos de sintonización basados en el principio del control LQR, el cual busca garantizar ciertas especificaciones por medio de una función de costo. Este enfoque se aplica a sistemas lineales con retardo de bajo orden (He *et al.*, 2000) y para sistemas de segundo orden con retardo (Srivastava *et al.*, 2016). Por último, hay técnicas donde se toman en cuenta las relaciones entre la sintonización del controlador PID y la utilización de los criterios de desempeño con base a la minimización de la integral del error (Lopez *et al.*, 1967; A. Rovira *et al.*, 1970). Al utilizar los índices de desempeño como función objetivo se obtiene una sintonización óptima del controlador (Martins, 2005).

A pesar de que los métodos anteriores pueden dar buenos resultados, no siempre es el caso. Por ello se han desarrollado nuevas heurísticas y metaheurísticas de optimización mono-objetivo y multi-objetivo, que en la práctica han dado mejores resultados en diferentes problemas complejos como los de ruta más corta (Di Caro y Dorigo, 1999; Ouaarab *et al.*, 2014) y asignación óptima (Liang y Smith, 2004). Una de estas técnicas es la programación cóncava-convexa que hace uso de restricciones

convexas (Hast *et al.*, 2013). Otras se basan en comportamientos de la naturaleza, como los algoritmos genéticos y evolución diferencial (Saad *et al.*, 2012), recocido simulado (Kirkpatrick *et al.*, 1983), explosión de mina (Majumdar *et al.*, 2014), y optimización por enjambre de partículas (Kennedy y Eberhart, 1995).

Dado que el cómputo bioinpirado ha permitido obtener buenos resultados en la solución de problemas de optimización compleja, surgió la idea de buscar y usar un método que en combinación con los índices de desempeño, y algunos criterios en el dominio del tiempo que permitiera obtener un conjunto de parámetros óptimos para el controlador PID.

1.2. Objetivos

1.2.1. Objetivo general

Implementar algoritmos bioinspirados para la sintonización automática de parámetros en controladores de sistemas con retardo.

1.2.2. Objetivos específicos

- Seleccionar modelos y controladores apropiados para el estudio.
- Seleccionar marco de desarrollo de programación.
- Seleccionar criterios de optimización.
- Seleccionar heurística de optimización y operadores correspondientes.
- Determinar el desempeño relativo entre las metaheurísticas.
- Ilustrar la propuesta de sintonización por medio de simulaciones.

1.3. Propuesta de solución

En este trabajo se propuso una metodología para sintonizar un controlador PID utilizando optimización tanto mono-objetivo como multi-objetivo. Los algoritmos que se utilizaron en el enfoque mono-objetivo son: evolución diferencial (DE, por sus siglas en inglés) y optimización por enjambre de partículas (PSO, por sus siglas en inglés)

(ver subsecciones 2.3.1 y 2.3.2). Como función objetivo se usaron los índices de desempeño (ver subsubsección 2.4.2.1), y parámetros de la forma de la respuesta, como el sobreimpulso y el tiempo de asentamiento (ver subsubsección 2.4.2.2). Estos son considerados porque ayudan a resaltar las especificaciones más importantes del sistema, como la respuesta en estado estacionario, el tiempo de asentamiento y el sobreimpulso.

En el enfoque multi-objetivo, se utilizó el algoritmo evolución diferencial para la optimización multi-objetivo (DEMO, por sus siglas en inglés) (ver subsección 2.3.3). Como funciones objetivo se utilizaron el sobreimpulso, el tiempo de asentamiento y el esfuerzo de control. Minimizándolas simultaneamente, se pretende encontrar un conjunto de soluciones óptimas que ofrezcan las mejores respuestas del sistema y además determinar si existe un grado de conflicto entre ellas.

Capítulo 2 Preliminares

En este capítulo se definen el concepto de retardo, los distintos sistemas dinámicos con retardo que se pueden utilizar y el modelo del sistema con retardo adoptado en este trabajo. Además se da una breve descripción del funcionamiento del controlador PID y de una de las técnicas clásicas que se utilizan para sintonizarlo. También se explican las metaheurísticas de optimización que fueron comparadas, los criterios de optimización que se tomaron en cuenta y las estrategias implementadas para llevar a cabo la optimización de la sintonización. Cabe mencionar que aunque este trabajo se enfoca en sistemas continuos con retardos, es posible aplicar la metodología usada a sistemas sin retardo. Se da por hecho que se conoce el modelo del sistema.

2.1. Sistemas con retardo

Un sistema con retardo o “tiempo muerto", es un sistema dinámico donde su evolución en el tiempo (estado futuro) no depende solamente de su estado actual, sino también de su historia pasada (Kolmanovskii y Myshkis, 1992).

Los sistemas con retardo se pueden clasificar de acuerdo a ciertas características (Kolmanovskii y Myshkis, 1992; Gu *et al.*, 2003):

- Por el argumento en su derivada de mayor grado.
  - Retardados: el sistema se puede modelar a partir de una ecuación diferencial funcional con retardo.
  - Neutrales: el sistema se puede modelar a partir de una ecuación diferencial funcional neutral.
- Por el número de retardos.
  - Único: el sistema tiene un solo retardo.

$$\dot{x}(t) = f(t, x(t), x(t-L)) \;\; L \in \mathbb{R}^+ \tag{1}$$

  - Múltiple: el sistema tiene más de un retardo.

$$\dot{x}(t) = f(t, x(t), x(t-L_1), x(t-L_2), ..., x(t-L_h)) \;\; L_i \in \mathbb{R}^+, i = 1, ..., h \tag{2}$$

- Por la ubicación del retardo.

- Entrada:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t-L) \\ y(t) &= Cx(t) \end{aligned} \tag{3}$$

- Salida:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t-L) \end{aligned} \tag{4}$$

- Estado:

$$\begin{aligned} \dot{x}(t) &= Ax(t-L) + Bu(t) \\ y(t) &= Cx(t) \end{aligned} \tag{5}$$

- Según la naturaleza del sistema:
  - Continuos o discretos: un sistema se considera continuo cuando sus variables son funciones que presentan continuidad en magnitud y en tiempo. Los sistemas discretos, presentan sucesiones o discontinuidades.
  - Lineales o no lineales: un sistema lineal es aquel que cumple con las propiedades de homogeneidad y superposición, caso contrario, es no lineal.
  - Determinista o estocástico: un sistema es determinista si sus variables son de esta naturaleza; un sistema es estocástico cuando al menos una de sus variables es aleatoria.
  - Una entrada y una salida, o múltiples entradas y múltiples salidas.

2.1.1. Modelo del sistema considerado

En este trabajo se utiliza un sistema de segundo orden con retardo. Este modelo se obtiene del trabajo presentado en Chen *et al.* (2017), el cual representa una bomba eléctrica de un sistema marino y se muestra a continuación.

$$G(s) = \frac{0.5e^{-0.2s}}{0.08s^2 + 0.68s + 1.45} \tag{6}$$

El sistema de segundo orden con retardo considerado se presenta en la ecuación (7) como función de transferencia, e incluye un operador e^{-sL}, que representa el retardo.

$$G(s) = \frac{Y(s)}{U(s)} = \frac{K\omega_n e^{-sL}}{s^2 + 2\xi\omega_n s + \omega_n^2} \tag{7}$$

También se utiliza la forma canónica de un sistema de segundo orden con retardo en la entrada dada en la ecuación (8).

$$\begin{aligned} \dot{x_1}(t) &= x_2(t) \\ \dot{x_2}(t) &= 2\xi\omega_n x_2(t) + \omega_n^2 x_1(t) + u(t-L) \\ y(t) &= x_1(t), \end{aligned} \tag{8}$$

donde $x \in \mathbb{R}^2$ es el vector de estado, $u(t) \in \mathbb{R}$ es la entrada del sistema y $y(t) \in \mathbb{R}$ la salida.

2.2. Estrategias de optimización

Para comparar la respuesta entre sistemas se utilizaron como función objetivo los índices de desempeño como se presenta en Dorf y Bishop (2011) y en Srivastava y Pandit (2016b). El problema de sintonización se aborda como un problema de optimización tanto mono-objetivo como multi-objetivo. Las formas convencionales de resolver problemas de optimización multiobjetivo son usar técnica de suma ponderada y la técnica basada en restricciones. En este trabajo se utilizó un tercer enfoque basado en la dominancia de Pareto.

2.2.1. Optimización mono-objetivo

Un problema de optimización mono-objetivo se presenta de la siguiente forma:

$$\begin{aligned} \text{minimizar} \quad & f(x) \\ \text{sujeta a} \quad & x \in \Omega. \end{aligned} \tag{9}$$

donde la función $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ es la que se desea minimizar y es llamada función objetivo o función de costo. $x = [x_1, x_2, ..., x_n]^T \in \mathbb{R}^n$ es un vector de n variables independientes. $x_1, ..., x_n$ conocidas como variables de decisión. El conjunto Ω es un subconjunto de $\mathbb{R}^n$ llamado conjunto de restricciones o región factible (Chong y Zak, 2013).

2.2.2. Optimización multi-objetivo

Un problema de optimización multi-objetivo se presenta de la siguiente forma:

$$\begin{aligned} \text{minimizar} \quad & F(x) = (f_1(x), f_2(x), ..., f_m(x))^T \\ \text{sujeta a} \quad & x \in \Omega. \end{aligned} \tag{10}$$

donde $F(x)$ consiste en m funciones objetivo $f_i : \mathbb{R}^n \longrightarrow \mathbb{R}$, $i = 1, 2, ..., m$; R^m es el espacio objetivo; Ω es la región factible y $x \in \Omega$ es el vector de decisión.

En este enfoque de optimización, mejorar una función objetivo dentro de $F(x)$, se puede lograr solo empeorando otra; esto significa que las funciones objetivo están en conflicto entre sí. En ese caso no hay una sola solución que optimice todos los objetivos simultáneamente. Sin embargo, al finalizar se tienen varias soluciones de compromiso (no dominadas), llamadas soluciones óptimas de Pareto, las cuales al graficarse en el espacio de las funciones objetivo conforman el frente de Pareto. En muchos problemas de optimización multiobjetivo es complejo obtener completamente y exactamente el conjunto de soluciones óptimas de Pareto, pero, es posible tener un aproximado al mismo (Von Lücken *et al.*, 2014).

El concepto de optimalidad de Pareto o dominancia fue presentado por primera vez por Edgeworth y Pareto (Pareto, 1906). Se establece que para dos vectores solución cualesquiera x^* y x existen cuatro posibles relaciones entre ellos.

1. La solución x^* domina a la solución x ($x^* \prec x$), si cada componente de $f(x^*)$ es igual o menor al componente correspondiente de $f(x)$, $f_i(x^*) \leq f_i(x) \, \forall i = 1, 2, ..., m$; y $f_i(x^*) \neq f_i(x)$ para al menos un i.
2. La solución x^* es dominada por la solución x ($x^* \succ x$). Para cada componente de $f(x^*)$ los componentes de $f(x)$ son iguales o mejores, $f_i(x^*) \geq f_i(x) \, \forall i = 1, 2, ..., m$; y $f_i(x^*) \neq f_i(x)$ para al menos un i.
3. La solución x^* es igual a la solución x ($x^* = x$). En cada uno de los componentes de $f(x^*)$, los componenetes correspondientes de $f(x)$ son los mismos, $f_i(x^*) = f_i(x) \, \forall i = 1, 2, .., m$.
4. La solución x^* es no comparable con la solución x ($x^* \succ\prec x$). Este caso ocurre cuando ninguno de los tres primeros casos sucede. en este caso se dice que ambas soluciones son igual de buenas (o igual de malas), $f_i(x^*) \leq f_i(x)$ y $f_j(x^*) \geq$

$f_j(x)$ para algún $i, j = 1,2,...,m$; y $x^* \neq x$.

La solución a un problema de optimización multi-objetivo es entonces el conjunto de soluciones no dominadas del conjunto de todas las soluciones factibles del problema. Este se denomina también conjunto de soluciones no-inferiores (Steuer, 1986) o decisiones Pareto óptimas (Pardalos *et al.*, 2017).

Definición 1. *Para un problema de optimización multi-objetivo, el conjunto óptimo de Pareto (P) se define como:*

$$P := \{x \in D | \nexists y \in D \, f(y) \prec f(x)\} \tag{11}$$

D es el conjunto de soluciones factibles.

Definición 2. *Para un problema de optimización multi-objetivo y un conjunto óptimo de Pareto (P), el frente Pareto se define como:*

$$PF := \{F(x) = [f_1(x), f_2(x), ..., f_m(x)]^T | x \in P\} \tag{12}$$

2.3. Metaheurísticas de optimización

Para resolver el problema de optimización y realizar la sintonización adecuada de los parámetros del controlador PID, se utilizaron tres algoritmos bioinspirados metaheurísticos: evolución diferencial, optimización por enjambre de partículas, ambos para enfoque mono-objetivo, y evolución diferencial para la optimización multi-objetivo (DEMO, por sus siglas en inglés). Estos métodos fueron seleccionados debido a que su implementación es relativamente sencilla, además no es necesario que al utilizar la función objetivo, esta deba cumplir con características específicas como diferenciabilidad, convexidad e inclusive continuidad, sólo es necesario obtener el valor de la función objetivo.

2.3.1. Evolución diferencial

Esta metaheurística fue presentada por Storn y Price (1997). Este método es estocástico y pertenece a la familia de los algoritmos evolutivos. Es muy similar a

los algoritmos genéticos, que están basados en los principios de la biología evolutiva. Este algoritmo realiza una búsqueda global en una población de posibles soluciones candidatas, que se mutan y combinan para crear nuevos individuos que serán seleccionados dependiendo de su valor de costo (aptitud) dado por la función objetivo. Una característica especial de este algoritmo es que utiliza vectores de prueba, los cuales compiten con cada uno de los individuos de la población actual con el fin de sobrevivir. El método se detiene cuando cumple con un número determinado de generaciones.

En el algoritmo evolución diferencial pueden adoptarse distintas estrategias dependiendo del tipo de problema que se esté abordando. Estas se clasifican con la notación $DE/x/y/z$, donde x especifica el vector objetivo a mutar, el cual puede ser "rand" (un vector de la población seleccionado aleatoriamente) o "best" (el vector con menor costo de la población) e y es el número de diferencia entre vectores. Por último, z indica el esquema de cruzamiento, el cual es cruce binomial "bin". En este trabajo se utilizó una variante llamada DE/rand/1/bin debido a que es la más recomendada y utilizada en la literatura.

Algoritmo

Para utilizar este método de búsqueda directa paralela debe generarse una población inicial aleatoria $\overline{NP} = \{1, ..., NP\}$ de vectores $x_{i,G} \in \mathbb{R}^D, i \in \overline{NP}, G \in \overline{G}$, dónde D es el número de parámetros involucrados en la función objetivo, $\overline{G} = \{1, ..., G_m\}$ es el número de generaciones y NP es igual a $10D$ o $15D$, por recomendación de los autores del método (Storn y Price, 1997).

El tamaño de la población nunca cambia durante el proceso de minimización. La aleatoridad con la que la población es creada, como regla general, se genera con una distribución de probabilidad uniforme. El algoritmo genera nuevos vectores de parámetros, llamados vectores mutados, agregando la diferencia ponderada entre dos vectores de la población a un tercer vector, todos seleccionados aleatoriamente. A esta operación se le conoce como mutación. Los parámetros de este vector mutado se combinan con los parámetros de un vector predeterminado, llamado vector objetivo, creando un nuevo vector, conocido como vector de prueba. Esta combinación se conoce como cruzamiento. Si el vector de prueba al ser evaluado en la función objetivo tiene un valor de costo menor al vector objetivo, este vector reemplaza al vector

objetivo en la siguiente generación. A esta operación se le conoce como selección. Cada vector de la población tiene que ser utilizado como vector objetivo para que las competiciones de los NP vectores tengan lugar en una generación. A continuación cada operación presente en el algoritmo se describe con más detalle.

Mutación

Para cada $x_{i,G} \in \mathbb{R}^D, i \in \overline{NP}$ se genera un vector mutado de acuerdo a la siguiente ecuación,

$$v_{i,G+1} = x_{a_1,G} + F(x_{a_2,G} - x_{a_3,G}), \tag{13}$$

donde $a_j \in \overline{NP}, a_j \neq i\ j = 1,2,3$ y $a_j \neq a_k\ j,k = 1,2,3, j < k$, son índices aleatorios enteros y $F > 0$. Los índices seleccionados al azar deben ser diferentes del índice i del vector objetivo. F es un factor constante y real $\in [0,2]$ que actúa como ganancia de la diferencia $(x_{a_2,G} - x_{a_3,G})$. En la Figura 1 se muestra un ejemplo tomado y re-dibujado de Storn y Price (1997), que presenta una función de costo bidimensional que ilustra los vectores involucrados en la generacion del vector mutado.

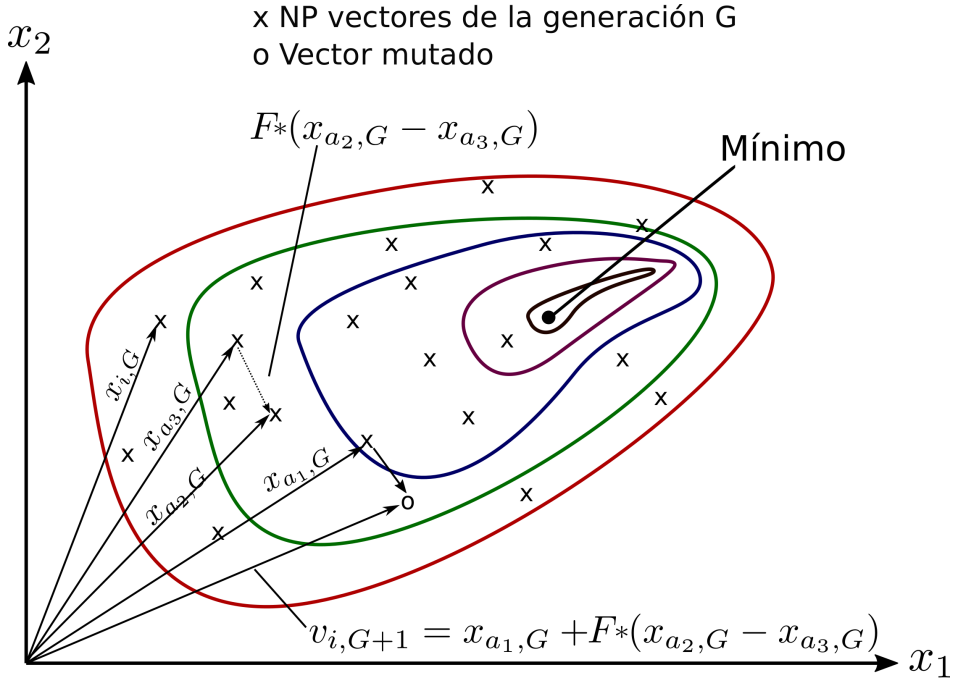

Figura 1. Función de costo bidimensional.

Cruzamiento

Para que exista más diversidad en los parámetros de los vectores mutados, se utiliza el cruzamiento, donde por medio de la ecuación (14) se crea el denominado vector de

prueba.

$$u_{ji,G+1} = \begin{cases} v_{ji,G+1} & \text{if } (randb(j) \leq CR) \text{ or } j = rnbr(i) \\ x_{ji,G} & \text{if } (randb(j) > CR) \end{cases}, \quad (14)$$
$$j = 1,2,...,D.$$

donde $u_{ji,G+1}$ es el vector de prueba que puede convertirse en miembro de la siguiente generación, $randb(j) \in [0,1]$ es la evaluación de un generador de números aleatorios uniforme. $CR \in [0,1]$ es la constante de cruzamiento, y $rnbr(i) \in [1,2,...,D]$ es un índice seleccionado al azar que asegura al menos un parámetro del vector mutado en el vector prueba.

En la Figura 2 se muestra un ejemplo de como se lleva a cabo este proceso. Se puede observar que al realizar el barrido en cada una de las posiciones correspondientes al tamaño de los vectores, en las posiciones uno, tres y cinco el vector de prueba heredó las características del vector objetivo, esto debido a que no se cumplió la primera condición expresada en la ecuación (14). Contrario a lo que ocurre en la posición dos y cuatro, que sí cumplen con la primera condición y por ende el vector de prueba heredó las características del vector mutado.

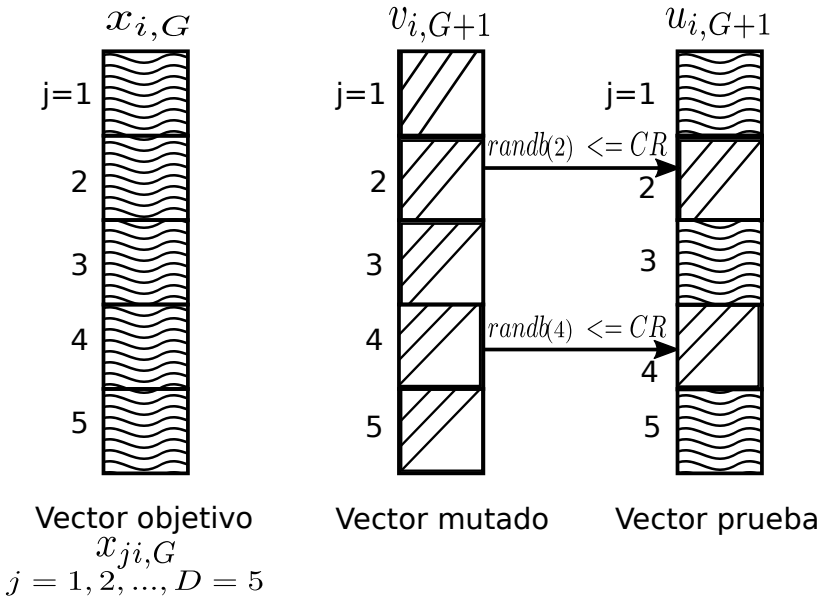

Figura 2. Proceso de cruzamiento.

Selección

Para decidir si el vector prueba $u_{i,G+1}$ debe convertirse en miembro de la generación $G+1$, se compara con el vector objetivo $x_{i,G}$, usando el criterio voraz, el cual consiste

en evaluar el vector $u_{i,G+1}$, y si se obtiene un valor en la función de costo menor al obtenido con $x_{i,G}$, entonces $x_{i,G+1}$ será $u_{i,G+1}$, si no, se conserva el vector objetivo $x_{i,G}$. Expuesto todo lo anterior, el pseudocódigo del método queda ilustrado en el Algoritmo 1.

Algoritmo 1: Pseudocódigo para el algoritmo evolución diferencial.

```
   Input: G, NP, S, F, CR
   Output: P_best
 1 begin
 2     Crear población inicial aleatoria x_{i,G} ∀i, i = 1,2,...,NP
 3     for g = 1 to G do
 4         for i = 1 to NP do
 5             Seleccionar aleatoriamente a_1 ≠ a_2 ≠ a_3 ≠ i
 6             Crear vector mutado: v_{i,G+1} = x_{a_1,G} + F(x_{a_2,G} − x_{a_3,G})
 7             for j = 1 to D do
 8                 if randb(j) ≤ CR or j = rnbr(i) then
 9                     u_{ji,G+1} = v_{ji,G+1}
10                 else
11                     u_{ji,G+1} = x_{ji,G}
12                 end
13             end
14             if f(u_{i,G+1}) ≤ f(x_{i,G}) then
15                 x_{i,G+1} = u_{i,G+1}
16             else
17                 x_{i,G+1} = x_{i,G}
18             end
19         end
20         G ← G + 1
21     end
22 end
```

2.3.2. Optimización por enjambre de partículas

Esta metaheurística fue presentada por Kennedy y Eberhart (1995). Este método al igual que evolución diferencial, es bioinspirado y tiene un enfoque estocástico, pertenece al campo de inteligencia de enjambre e inteligencia colectiva. La inspiración de su desarrollo surgió a raíz de la elaboración de modelos de conductas sociales de los patrones de movimiento del vuelo de las aves y de los bancos de peces. Comparte similitudes con los métodos de computación evolutiva, ya que se inicializa con una población de soluciones candidatas aleatorias y busca el valor óptimo al avanzar en las generaciones, aunque no utiliza los operadores de mutación y cruzamiento.

En este algoritmo a las soluciones candidatas se les llama "partículas", las cuales están en constante movimiento en un espacio de búsqueda multidimensional de un problema de optimización siguiendo a las partículas óptimas actuales. La posición de cada partícula representa una solución candidata al problema en cuestión, y cada partícula busca estar en una mejor posición en el espacio de búsqueda al cambiar su velocidad, la cual está compuesta por una velocidad aleatoria inicial y la influencia ponderada de cinco parámetros: dos términos aleatorios $r_1, r_2 \in [0,1]$, las constantes cognitiva $c_1 \in \mathbb{R}^+$, social $c_2 \in \mathbb{R}^+$ y de inercia $w \in \mathbb{R}^+$.

Algoritmo

Para utilizar este método debe generarse una población inicial de vectores (partículas) de dimensión NP.

$$P_i, i = 1,2,...,NP \tag{15}$$

Cada partícula será inicializada con una posición ($p_i \in \mathbb{R}^n$) aleatoria uniformemente distribuida, utilizando como valores máximos y mínimos, las respectivas cotas inferiores y superiores del espacio de búsqueda de cada parámetro involucrado en el problema de optimización. También se inicializa la velocidad ($v_i \in \mathbb{R}^n$) en cero, aunque existe otra alternativa donde se establece de forma aleatoria uniformemente distribuida entre 0 y 1, o utilizando los límites del espacio de búsqueda. Después de inicializar, se crea un memoria de la mejor posición conocida de la partícula a su posición inicial ($p_{best_i} = p_i$) y evaluando la posición de cada partícula de la población $f(p_i)$ se encuentra la mejor partícula mediante $f(p_i) < f(g_{best})$, donde g_{best} es la mejor posición global conocida, y se establece $g_{best} = p_i$. Hecho esto, se realiza un proceso iterativo que mientras no se cumpla un criterio de paro (por ejemplo, máximo de iteraciones o solución satisfactoria) actualizará la velocidad y la posición de las partículas. La velocidad se actualiza mediante la ecuación que se muestra a continuación,

$$v_i^{k+1} = w * v_i^k + c_1 * r_1 * (p_{best_i} - p_i^k) + c_2 * r_2 * (g_{best} - p_i^k). \tag{16}$$

Con esta nueva velocidad se actualiza la posición,

$$p_i^{k+1} = p_i^k + v_i^{k+1}. \tag{17}$$

De nuevo se evalúa cada partícula y si $f(p_i) < f(p_{best_i})$ se actualiza p_{best_i} y si $f(p_{best_i}) <$

$f(g_{best})$ se actualiza g_{best}. El algoritmo muestra g_{best} hasta cumplir la condición de paro.

Un ejemplo tridimensional del comportamiento que tienen las partículas durante el proceso se puede ver en la Figura 3 y el proceso explicado de forma gráfica por partícula puede verse en la Figura 4. Expuesto todo lo anterior, el pseudocódigo del método queda ilustrado en el Algoritmo 2.

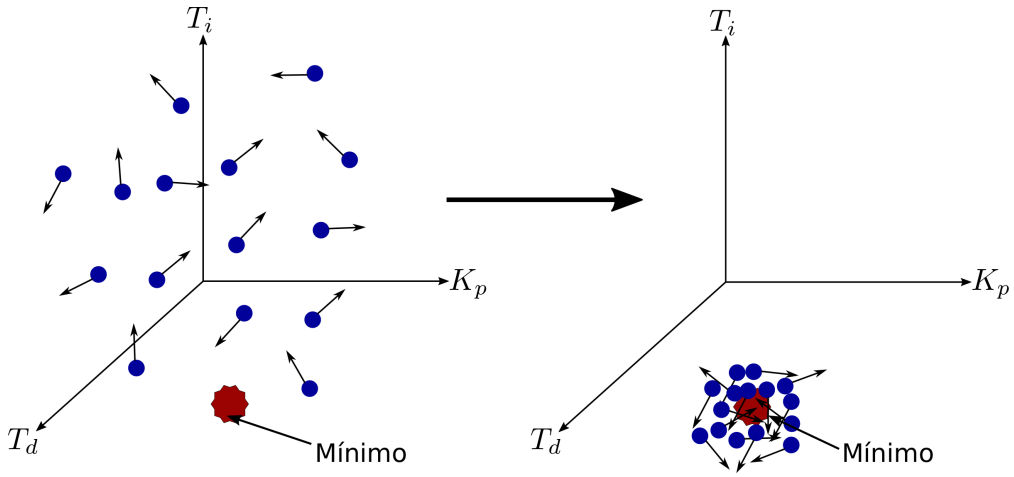

Figura 3. Ejemplo tridimensional del movimiento de las partículas: proceso con cero iteraciones (izquierda), después de N iteraciones (derecha).

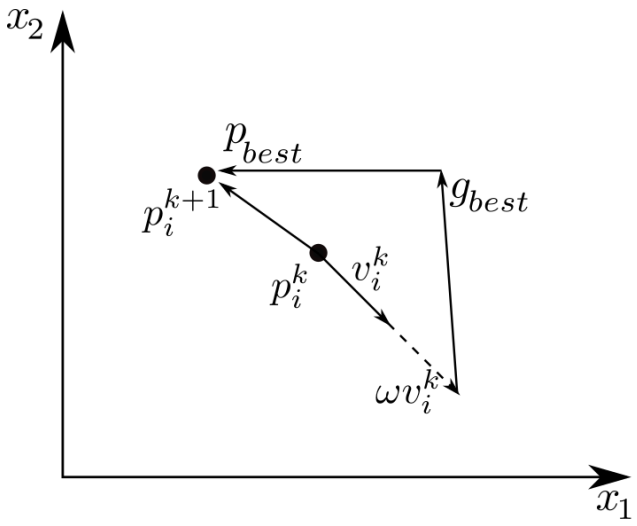

Figura 4. Ejemplo bidimiensional del movimiento de la partícula en el espacio de búsqueda.

2.3.3. Evolución diferencial para la optimización multi-objetivo

Esta metaheurística fue presentada por Robič y Filipič (2005). Es un algoritmo que aprovecha y combina las ventajas que ofrece evolución diferencial con los mecanismos de clasificación basada en la dominancia de Pareto, y la clasificación de

Algoritmo 2: Pseudocódigo para el algoritmo de optimización por enjambre de partículas.

Input: I, NP, S
Output: g_{best}

begin
  $P \longleftarrow P_0$
  $g_{best} \longleftarrow 0$
  for $i = 1$ **to** NP **do**
    $p \longleftarrow$ Posición aleatoria
    $v \longleftarrow 0$
    $p_{best_i} \longleftarrow p_i$
    if $f(p_{best_i}) \leq f(g_{best})$ **then**
      $g_{best} \longleftarrow p_{best_i}$
    else
    end
  end
  while $k < NP + 1$ **do**
    for j **to** NP **do**
      $v \longleftarrow v_i^{k+1} = w * v_i^k + c_1 * r_1 * (p_{best_i} - p_i^k) + c_2 * r_2 * (g_{best} - p_i^k)$
      $p \longleftarrow p_i^{k+1} = p_i^k + v_i^{k+1}$
      if $f(p) \leq f(p_{best})$ **then**
        $p_{best} \longleftarrow p$
        if $f(p_{best}) \leq f(g_{best})$ **then**
          $g_{best} \longleftarrow p_{best}$
        else
        end
      else
      end
    end
  end
end

distancia de amontonamiento. Puede implementarse con tres variantes, DEMO/parent, DEMO/closest/dec y DEMO/closest/obj. Estas dos últimas utilizan el concepto de amontonamiento (“crowding") y se diferencian en que una es comparada con algún individuo similar dentro del espacio de decisión, y la otra compara con algún individuo similar dentro del espacio de decisión y del espacio objetivo.

En este trabajo se utilizó la variante DEMO/parent la cual sigue las siguientes reglas:

1. El candidato reemplaza al padre si lo domina.

2. Si el padre domina al candidato, el candidato se descarta.

3. Cuando ninguno de los dos se dominan entre si, el candidato se agrega a la población.

Esto se repite hasta que se crean NP (tamaño de la población) candidatos. Si la población final es mayor a NP, se debe truncar. La truncación consiste en clasificar a los individuos con clasificación no dominada y luego evaluar a los individuos de un mismo frente con la métrica de distancia de amontonamiento, la cual consiste en seleccionar a los individuos más distanciados. Este proceso de truncamiento mantiene a los NP mejores individuos. El truncamiento se deriva del algoritmo de optimización multi-objetivo NSGA-II (Deb *et al.*, 2002).

2.4. Criterios de optimización

Para definir una función objetivo se pueden tomar en cuenta desde uno hasta varios criterios. Estos criterios se pueden dividir en dos categorías: criterios en el dominio de la frecuencia y criterios en el dominio del tiempo. Estos últimos, en los trabajos existentes en la literatura son los criterios más utilizados (Sahib y Ahmed, 2016).

2.4.1. Criterios en el dominio de la frecuencia

Estos criterios por lo general están relacionados con la función de transferencia en lazo cerrado del sistema de control, algunos de ellos son los siguientes:

- Frecuencia de cruce de ganancia ω_{gc}.
- Margen de ganancia g_m.
- Margen de fase ϕ_m.
- Máxima sensibilidad M_s.
- Frecuencia donde la función de sensibilidad tiene su máximo ω_{ms}.
- Frecuencia de cruce de sensibilidad ω_{sc}.
- Sensibilidad complementaria máxima M_t.
- Frecuencia donde la función de sensibilidad complementaria tiene su máximo ω_{mt}.

Algoritmo 3: Pseudocódigo para el algoritmo de evolución diferencial para la optimización multiobjetivo.

Input: G, NP, S, F, CR
Output: P_{best}

begin
  Crear población inicial P aleatoria $x_{i,G}\ \forall i, i = 1,2,...,NP$
  Evaluar población inicial $x_{i,G}\ \forall i, i = 1,2,...,NP$
  for $g = 1$ **to** G **do**
    for $i = 1$ **to** NP **do**
      Seleccionar aleatoriamente $a_1 \neq a_2 \neq a_3 \neq i$
      Crear vector mutado: $v_{i,G+1} = x_{a_1,G} + F(x_{a_2,G} - x_{a_3,G})$
      for $j = 1$ **to** D **do**
        if $randb(j) \leq CR$ or $j = rnbr(i)$ **then**
          $u_{ji,G+1} = v_{ji,G+1}$
        else
          $u_{ji,G+1} = x_{ji,G}$
        end
      end
      if $f(u_{i,G+1}) \prec f(x_{i,G})$ **then**
        $x_{i,G+1} = u_{i,G+1}$
      else
        if $f(x_{i,G}) \prec f(u_{i,G+1})$ **then**
        else
          $NP = NP \cup \{u_{i,G+1}\}$
        end
      end
      if $P > NP$ **then**
        Truncar
      else
      end
      Enumerar aleatoriamente los individuos en NP
    end
    $G \leftarrow G + 1$
  end
end

2.4.2. Criterios en el dominio del tiempo

Estos criterios se basan en la respuesta del sistema a una señal escalón en la entrada. Se puede dividir en dos categorías:

- Índices de desempeño.
- Parámetros de la forma de la respuesta.

2.4.2.1. Índices de desempeño

En Dorf y Bishop (2011) se menciona que un índice de desempeño es una medida cuantitativa que sirve para evaluar el desempeño de un sistema y se elige de forma que resalte las especificaciones más importantes del sistema. Dado esto, el mejor sistema es aquel que minimice o tenga el valor menor en dicho índice.

Los índices de desempeño se calculan con base a un tiempo finito elegido de forma arbitraria T, de forma que el sistema alcance su estado estacionario. Es conveniente elegir T como el tiempo de asentamiento (t_s), esto con propósito computacional (Rao, 1993).

Los índices de desempeño se obtienen en función del error, definido por la diferencia entre la referencia del sistema y la salida del sistema, como se muestra en la siguiente ecuación:

$$e(t) = y_{ref}(t) - y(t).$$

Los índices utilizados se describen a continuación. Todos son nombrados por sus siglas en inglés.

IAE

El IAE, o integral del valor absoluto del error, que se define como

$$IAE = \int_0^\infty |e(t)| dt, \tag{18}$$

solamente va acumulando el error presente en el transitorio sin añadir peso a algún error, no cancela los cambios de signo, da una respuesta bien amortiguada y con muy poco sobrepaso (Shinskey, 1996). En Dorf y Bishop (2011) se menciona que es particularmente útil para estudios de simulación.

IE

El IE, o integral del error, que se define como

$$IE = \int_0^\infty e(t) dt, \tag{19}$$

es igual al IAE cuando el error no cambia de signo (Åström y Hägglund, 2006) y

es una buena aproximación a IAE cuando se tiene un sistema oscilatorio pero bien amortiguado (Åström y Hägglund, 1995).

ISE

El ISE, o integral del error cuadrático, que se define como

$$ISE = \int_0^\infty e^2(t)dt, \tag{20}$$

dará siempre un resultado positivo, esto debido a que la función de error está al cuadrado, lo que hace que se penalicen valores tanto positivos como negativos (Shinners, 1998). Este criterio da mayor peso a errores grandes pero da muy poco peso a errores pequeños (Ogata, 1990). Es fácil de calcular tanto analíticamente como computacionalmente y ayuda a discernir entre sistemas excesivamente subamortiguados y excesivamente sobreamortiguados (Dorf y Bishop, 2011).

ITAE

El ITAE (Graham y Lathrop, 1953), o integral del tiempo multiplicado por el valor absoluto del error, que se define como

$$ITAE = \int_0^\infty t|e(t)|dt, \tag{21}$$

al ponderar el valor absoluto del error con el tiempo, los errores grandes presentes al inicio pasan desapercibidos ya que se multiplican con tiempos muy pequeños, esto quiere decir que da poco peso a los errores grandes al inicio y da un mayor peso a los errores pequeños presentes a lo largo del tiempo. Genera sobrepaso y oscilaciones más pequeñas que el IAE y el ISE, pero es complicado de calcular analíticamente (Ogata, 1990).

ITSE

El ITSE, o integral del tiempo multiplicado por el error cuadrático, que se define como

$$ITSE = \int_0^\infty te^2(t)dt, \tag{22}$$

otorga poco peso a los errores iniciales, pero penaliza fuertemente los errores presentes en un tiempo posterior en la respuesta transitoria (Ogata, 1990).

ISTAE

El ISTAE, o integral del tiempo al cuadrado multiplicado por el error absoluto, que se define como

$$ISTAE = \int_0^{\infty} t^2 |e(t)| dt, \tag{23}$$

otorga poco peso a los errores grandes al inicio de presentarse una entrada escalón en el sistema, y da mayor peso a los errores presentes en un tiempo posterior a la aplicación del escalón.

ISTSE

El ISTSE, o integral del tiempo al cuadrado multiplicado por el error cuadrático, que se define como

$$ISTSE = \int_0^{\infty} t^2 e(t)^2 dt, \tag{24}$$

disminuye de forma rápida los errores grandes presentes al inicio de la respuesta transitoria.

2.4.2.2. Parámetros de la forma de la respuesta

Sobreimpulso

El sobreimpulso (M_p), se define como el pico máximo de la respuesta del sistema (ver Figura 5).

Tiempo de asentamiento

El tiempo de asentamiento (t_s), se define como el tiempo a partir del cual la curva de la respuesta se mantiene dentro de un intervalo usualmente del 2% o del 5% del valor en estado estacionario (ver Figura 5).

2.5. Controlador PID

El control conocido como PID contiene una acción proporcional, una integral y una derivativa. De esta forma aprovecha las ventajas y elimina las desventajas que cada una tiene si se utilizan por si solas. Trabaja con base al error ($e(t)$) existente entre una

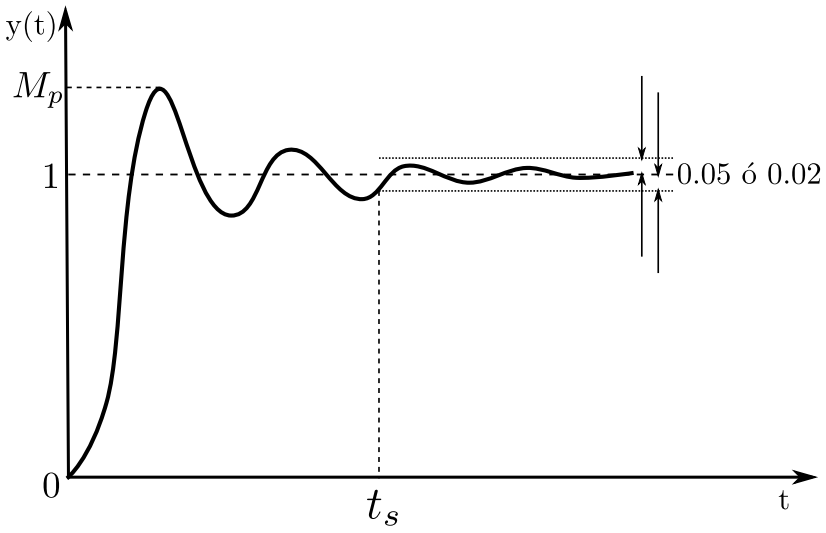

Figura 5. Características de una respuesta al escalón.

referencia ($y_{ref}(t)$) y la salida ($y(t)$) de la planta, es decir,

$$e(t) = y_{ref}(t) - y(t). \tag{25}$$

Un esquema básico en el que se suele implementar el controlador se presenta en la Figura 6.

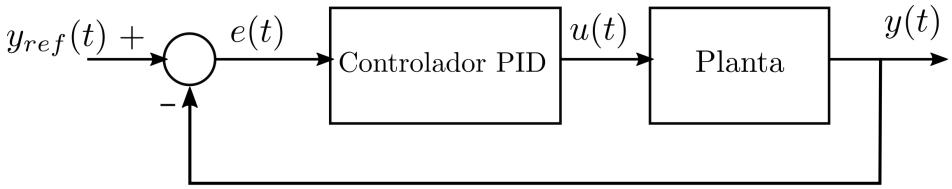

Figura 6. Esquema básico de implementación del controlador PID.

La ley de control en el tiempo $u(t)$, que combina las tres acciones de control es

$$u(t) = K_p e(t) + \frac{K_p}{T_i}\int_0^t e(\tau)d\tau + K_p T_d \frac{d}{dt}e(t), \tag{26}$$

sin embargo, no siempre se aplica en la forma completa. Por ello existe la forma clásica donde la acción proporcional afecta a las ganancias de las acciones restantes, y se representa en el dominio del tiempo de la siguiente forma

$$u(t) = K_p\Big(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d \frac{d}{dt}e(t)\Big), \tag{27}$$

donde K_p es la ganancia propocional, T_i el tiempo integral y T_d el tiempo derivativo.

En el dominio de la frecuencia se representa de la siguiente manera

$$U(s) = K_p\left(1 + \frac{1}{sT_i} + sT_d\right) \tag{28}$$

Acción proporcional

La acción proporcional está representada por el primer término de la ecuación (26). Un control proporcional puro, es de la forma

$$u(t) = K_p e(t),$$

donde el control básicamente es proporcional al error. Este tipo de control responde inmediatamente cuando se produce el error, pero si el error es relativamente pequeño tiene la desventaja de que produce un error en estado estacionario distinto a cero.

Acción integral

La acción integral se tiene en el segundo término de la ecuación 26

$$u(t) = \frac{K_p}{T_i}\int_0^t e(\tau)d\tau,$$

y su función primordial es hacer que el error en estado estable provocado por la acción proporcional sea igual a cero, es decir, que la salida de la planta tenga el mismo valor que la referencia. Esto lo logra acumulando el error conforme pasa el tiempo, lo que incrementa la acción integral hasta que la magnitud del control tenga efecto en la planta y el error se haga cero.

Acción derivativa

La acción derivativa se obtiene del último término de la ecuación (26)

$$u(t) = K_p T_d \frac{d}{dt}e(t)$$

y su función principal es mejorar la respuesta del sistema en lazo cerrado, actuando inmediatamente cuando ocurre un error sin que este llegue a tener un valor considerable, en otros términos se puede decir que “adelanta” al error. La desventaja que presenta, es que si hay exceso de ruido en la señal de medición (salida), el ruido se amplifica.

2.6. Sintonización Ziegler-Nichols

Los métodos o heurísticas de sintonización clásicos para determinar los parámetros del control PID fueron presentados por Ziegler y Nichols (1942), han tenido un gran impacto en la práctica y actualmente aún es utilizado, ya no tanto en la aplicación pero si como base y referencia de comparación, ya que no otorgan una buena sintonía. Estos métodos se basan en la caracterización de la dinámica del proceso mediante dos parámetros con los cuales se determinan los parámetros del controlador PID.

2.6.1. Primer método de Ziegler-Nichols

Conocido también como método de la respuesta en escalón, se basa en obtener mediante la aplicación de una entrada escalón al sistema en lazo abierto información de dos parámetros: a y L. Para obtener estos parámetros, primero se determina el punto donde la pendiente de la respuesta tiene su mayor valor y se dibuja la recta tangente a ese punto. Las intersecciones que conforma dicha recta con los ejes de coordenadas determinan los parámetros a y L. Es indispensable que la respuesta del sistema presente una forma de "S", como se puede apreciar en la Figura 7, y que el sistema no tenga integradores o polos dominantes complejos conjugados, para que este método se pueda aplicar. Ya que se conocen los valores de a y L, los parámetros del controlador PID se obtienen con base a la Tabla 1.

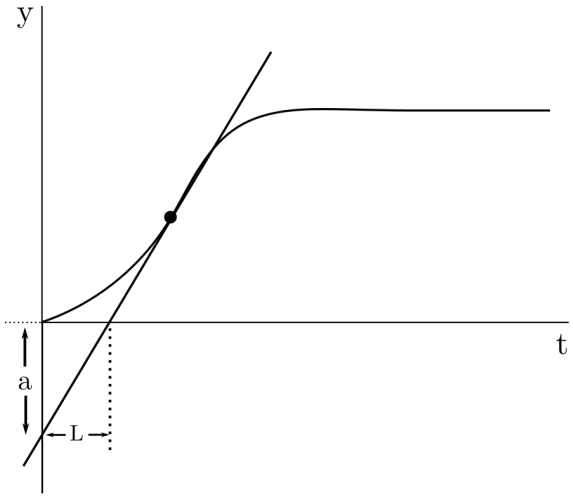

Figura 7. Caracterización de la respuesta en escalón en el método de la respuesta a un escalón.

Tabla 1. Parámetros del controlador para el método de la respuesta a un escalón.

Controlador	K_p	T_i	T_d
P	$\frac{1}{a}$		
PI	$\frac{0.9}{a}$	$3L$	
PID	$\frac{1.2}{a}$	$2L$	$\frac{L}{2}$

2.6.2. Segundo método de Ziegler-Nichols

Conocido también como método de la respuesta en frecuencia, igual que en el método anterior, se realiza una caracterización de la dinámica del sistema, sólo que ahora el sistema se pone en lazo cerrado utilizando nada más la acción de control proporcional. Una vez hecho esto, se debe incrementar K_p, empezando de cero, hasta que el sistema presente oscilaciones sostenidas en la salida. Esto lleva al sistema al límite de su estabilidad. Los parámetros que realizan la caracterización de la dinámica del sistema son K_u y T_u, donde K_u es la ganancia proporcional que hace oscilar el sistema de manera sostenida, con período de oscilación T_u. Una vez determinados estos parámetros, los parámetros del PID se obtienen con la Tabla 2.

Tabla 2. Parámetros del controlador para el método de la respuesta en frecuencia.

Controlador	K_p	T_i	T_d
P	$0.5K_u$		
PI	$0.4K_u$	$0.8T_u$	
PID	$0.6K_u$	$0.5T_u$	$0.125T_u$

Capítulo 3 Metodología propuesta

En este capítulo se presenta la metodología para lograr la sintonización de los parámetros del controlador PID. Para ello, se establecieron los enfoques de un problema de optimización mono-objetivo y multi-objetivo. Para el caso mono-objetivo se definió una función objetivo a minimizar, siendo esta un índice de desempeño y dos criterios en el dominio del tiempo, el sobreimpulso y el tiempo de asentamiento. Para la estrategia multi-objetivo se utilizaron los criterios de sobreimpulso (M_p), tiempo de asentamiento (t_s) y esfuerzo de control ($||u||_\infty$). El problema se resolvió mediante simulaciones numéricas.

3.1. Simulación

La sintonización del controlador PID se llevó a cabo por medio de la implementación de las dos metaheurísticas de optimización codificadas en lenguaje interpretado Python, tomando como motivación lo presentado por Rooy (2017a,b), la simulación de los sistemas con retardo se realizó mediante código en lenguaje C/C++ y el apoyo de una biblioteca para Python llamada pydelay (Flunkert y Schoell, 2009).

En la Figura 8 se muestra un diagrama de flujo de los algoritmos que se llevan a cabo para la sintonización del controlador con optimización mono-objetivo y multi-objetivo, y en la Figura 9 se muestra un diagrama de bloques general de la propuesta para sintonizar el controlador PID. Existen ciertas diferencias en la implementación tanto de los algoritmos como de los enfoques de optimización, y en los siguientes apartados se proporciona un poco más de detalle sobre cada uno.

3.1.1. Parámetros iniciales

Como se observó en las subsecciones 2.3.1 y 2.3.2 cada algoritmo utiliza ciertos parámetros que deben establecerse antes de realizar las simulaciones para obtener resultados adecuados. Cabe mencionar que estos parámetros fueron seleccionados según las recomendaciones publicadas (Storn y Price, 1997; Kennedy y Eberhart, 1995; Chen *et al.*, 2017). Estos parámetros se presentan en la subsección 4.1.1.

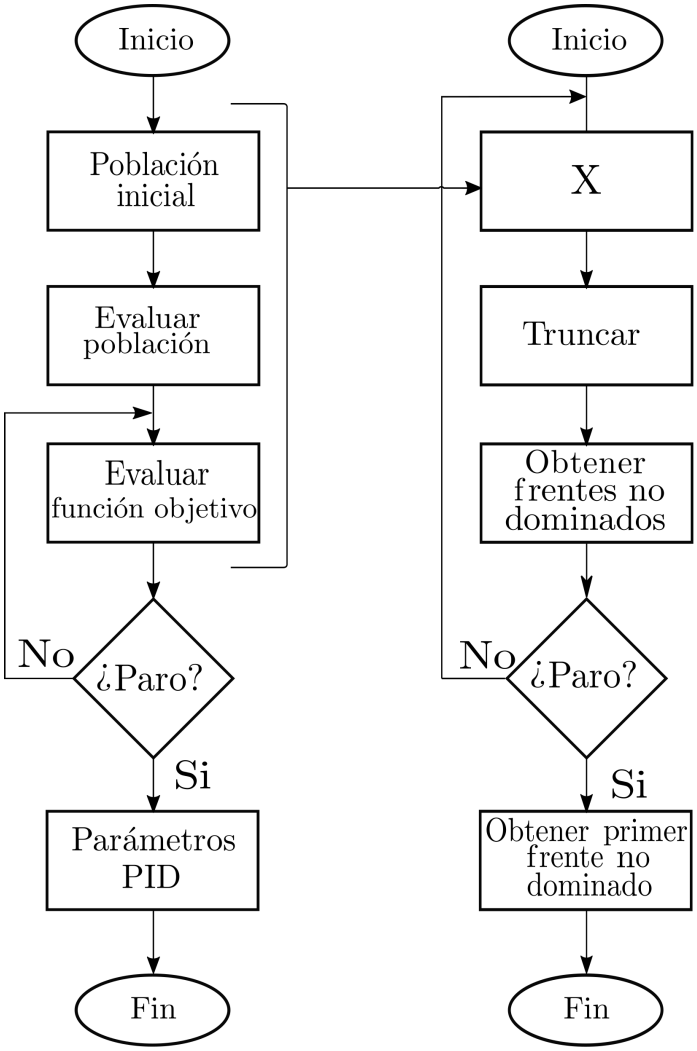

Figura 8. Diagrama de flujo de la sintonización del controlador PID con enfoque mono-objetivo (izquierda) y multi-objetivo (derecha).

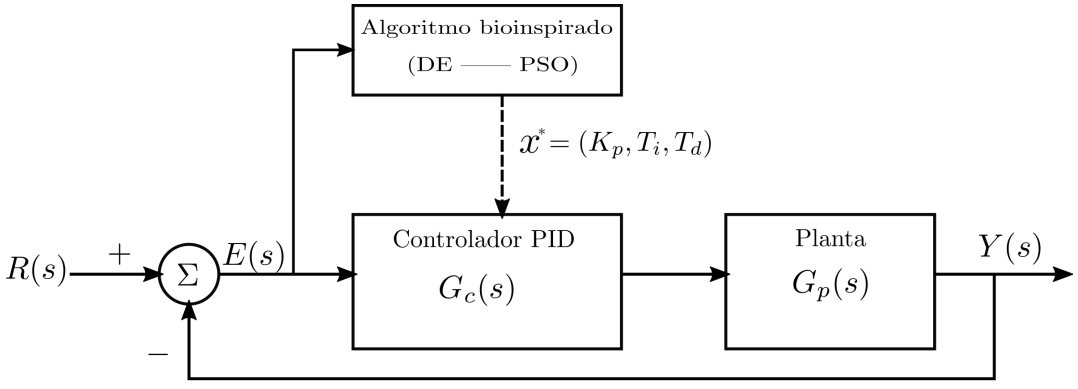

Figura 9. Diagrama de bloques de la sintonización de un controlador PID

3.1.2. Población inicial

Dependiendo de la metaheurística de optimización, se generará una población inicial a partir de un espacio de búsqueda determinado por los parámetros que se necesiten sintonizar en el controlador. Dicho espacio de búsqueda para el controlador PID queda representado de la siguiente forma, $S = \left[[X_{1_{min}}, X_{1_{max}}], [X_{2_{min}}, X_{2_{max}}], [X_{3_{min}}, X_{3_{max}}]\right]$, donde $X_i, i = 1,2,3$ son los parámetros K_p, T_i y T_d, respectivamente.

La mejor forma de representar los intervalos presentes en el espacio de búsqueda, es utilizando la heurística de Z-N. Con dicha técnica se obtienen los parámetros del controlador PID, que se utilizan como semilla para este espacio. En este trabajo por cuestiones de practicidad en las simulaciones y por realizar comparaciones justas, se optó por utilizar el espacio de búsqueda propuesto en Chen *et al.* (2017).

La población es generada mediante un proceso aleatorio por lo cual será distinta en los dos métodos, inclusive en cada simulación se tendrá una población inicial distinta. Para la metaheurística DE, la población es creada mediante la implementación de una distribución de probabilidad uniforme utilizando como límites superior e inferior los intervalos presentes en el espacio de búsqueda.

Para PSO, la población o enjambre de partículas es inicializada con dos datos importantes, la posición y la velocidad; la posición es aleatoria con una probabilidad uniformemente distribuida con base al espacio de búsqueda, igual que en el caso de evolución diferencial; y la velocidad, en este caso, se optó por utilizar la variante en donde se inicializa en cero para cada una de las partículas presentes en el cúmulo.

Para la metaheurística DEMO, la población inicial es generada igual que en el método DE utilizada en el enfoque mono-objetivo.

3.1.3. Evaluar población

En este bloque los algoritmos básicamente lo que hacen es obtener la aptitud (fitness) o costo de cada uno de los individuos en el caso de DE y DEMO, y la posición de las partículas en el caso de PSO. Además del costo se obtienen otras características propias como el sobreimpulso, el tiempo de asentamiento y las normas 1, 2 e infinito

de la ley de control.

3.1.4. Evaluar función objetivo

En este bloque ocurren varias acciones en cada algoritmo que se muestran explícitamente en el diagrama de flujo de la Figura 10. A continuación se explican los procesos presentes en este bloque.

En la metaheurística DE, antes de evaluar los parámetros en discusión se debe pasar por varios procesos (mencionados en la subsección 2.3.1). Después de esto, los vectores que contienen un conjunto de paŕametros del PID (vector mutado y objetivo), se envían por medio de un túnel de datos a un subprograma codificado en lenguaje C++. En este subprograma se lleva a cabo la simulación del sistema con retardo, donde se obtiene la respuesta del sistema y también se calcula el valor de la función objetivo (criterios de optimización). En total se tienen nueve criterios de optimización, pero en el código del programa en Python se selecciona qué criterio se desea optimizar. A la par del cálculo del criterio de optimización también se obtiene el valor del sobreimpulso, el tiempo de asentamiento y las normas 1, 2 e infinito de la ley de control. Además como se desea obtener la mejor respuesta, se utilizaron restricciones para forzar al algoritmo a tener un mejor conjunto de parámetros PID; una de las restricciones, es el sobreimpulso, donde se estableció $M_{po} < M_p$, siendo M_{po} el sobreimpulso del vector evaluado y M_p el menor sobreimpulso de la población; otra restricción que se utilizó fue el tiempo de asentamiento, $t_{so} < t_s$, donde, t_{so} es el tiempo de asentamiento del vector evaluado y t_s, el menor tiempo de asentamiento de la población.

Al evaluar las partículas, en PSO se envía por el túnel de datos al subprograma en lenguaje C++ cada uno de los conjuntos de parámetros del PID y se calcula, como en el caso anterior, la respuesta del sistema y el valor del criterio de optimización y se aplican las mismas restricciones de sobreimpulso y tiempo de asentamiento.

En el algoritmo de evolución diferencial para optimización multi-objetivo ocurre lo mismo que en la variante mono-objetivo, de igual forma los conjuntos de parámetros PID son enviados a un subprograma donde se obtiene el valor de la función de costo y la respuesta del sistema sujeto a las restricciones mencionadas anteriormente.

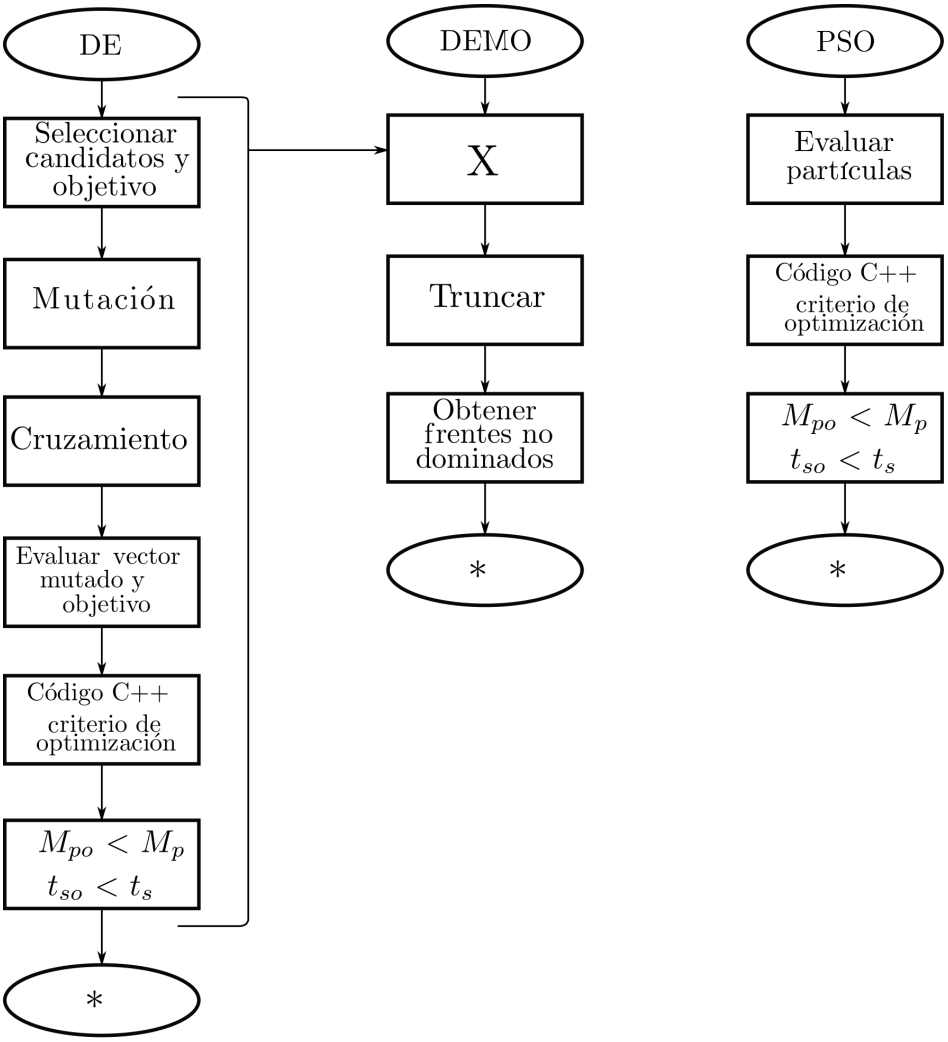

Figura 10. Diagramas de flujo del bloque para evaluar función objetivo de cada algoritmo.

3.1.5. Paro

La sintonización termina cuando se cumple el número de generaciones (en el caso de DE) y de iteraciones (en el caso de PSO) establecidas.

3.1.6. Parámetros del PID

Dependiendo del algoritmo de optimización, los nuevos parámetros del PID son seleccionados, según lo mostrado en la sección 2.3. En adición a lo expuesto en la sección mencionada, cuando se verifica, en el caso de evolución diferencial, que el vector mutado es mejor que el vector objetivo, también se verifica que dicho vector cumpla con las restricciones de sobreimpulso y tiempo de asentamiento. Si no se cumplen, el vector objetivo es el que se selecciona. En el caso de PSO, una vez que se

tiene el mejor vector global, de igual forma se verifica que cumpla con las restricciones impuestas.

Para el caso multi-objetivo la selección cambia, ya que se seleccionan los treinta mejores vectores que cumplen con las restricciones y con base a ellos se obtiene el frente no dominado de soluciones.

3.1.7. Truncar y obtener frentes no dominados

En este bloque correspondiente a la optimización multi-objetivo se verifica que al finalizar una generación, el número total de individuos en la población sea mayor al tamaño original de la misma. Si es así, se realiza un proceso donde se seleccionan los treinta mejores individuos presentes en los frentes aproximados de Pareto, y se crea una nueva población, la cual posteriormente es acomodada aleatoriamente.

Una vez que termina el algoritmo de iterar el número de generaciones, se tiene un conjunto de soluciones que conforman frentes aproximados de pareto. De estos frentes se seleciona el primer frente no dominado y ese es el conjunto de soluciones óptimas que se utilizan para sintonizar el controlador PID.

Capítulo 4 Experimentos y Resultados

En este capítulo se presentan y comparan los resultados obtenidos en simulación de la implementación de las técnicas evolutivas para la sintonización óptima de los parámetros del controlador PID. En todas las ejecuciones, la sintonización se llevó a cabo evaluando la respuesta del sistema ante un escalón unitario.

Para la simulación se utilizó el sistema de segundo orden con retardo en la entrada presentado en la subsección 2.1.1. El tiempo de simulación para cada combinación de parámetros es de 25 segundos.

A continuación se presentan primero las configuraciones de hardware y software utilizados para llevar a cabo las simulaciones, después los parámetros utilizados en cada metaheurística y por último los resultados obtenidos con DE, PSO y DEMO para cada enfoque de optimización.

4.1. Especificaciones de hardware y software

Hardware

Para las simulaciones de la sintonización del controlador PID se utilizó una computadora con las siguientes especificaciones:

- Procesador: Intel(R) Core(TM) i7-4790K CPU 4.00GHz
- Memoria: 16Gb
- SO: Ubuntu 16.04.2 LTS

Software

Para realizar la sintonización se utilizó el lenguaje de programación Python en conjunto con C++, con la finalidad de hacer más veloces y eficientes las simulaciones.

En lenguaje Python se realizaron las siguientes rutinas:

- Desarrollo de los algoritmos DE, PSO y DEMO (ver sección A.1).
- Simulación de los sistema de control con la biblioteca pydelay (ver subsección A.1.3).
- Técnicas de probabilidad y estadística.

En lenguaje C/C++ se realizaron las siguientes rutinas:

- Simulación de los sistemas de control (ver sección B.1).
- Obtención de la respuesta del sistema y ley de control.
- Cálculo de la función objetivo.
- Cálculo de las normas 1,2 e infinito para la ley de control.
- Cálculo del sobreimpulso.
- Cálculo del tiempo de asentamiento.

Las especificaciones del software utilizado son las siguientes:

- Python 2.7.12.
- Bibliotecas: pydelay 0.1.1, pygmo 2.8, subprocess 3.5.2, numpy 1.15.1, scipy 1.1.0, pylab 2.2.3.
- Compilador gcc 5.5.0.

Python ofrece las siguientes ventajas (Oliphant, 2007; Pérez y Granger, 2007):

- Amplia biblioteca para programación científica.
- Sintaxis clara (legibilidad).
- Capacidad de interactuar con otros softwares.
- Facilidad de uso.

Estas características conllevan que tenga la desventaja de que sea un lenguaje de programación con ejecución lenta, pero al combinarse con programas en C++ se puede aprovechar lo mejor de ambos lenguajes.

4.1.1. Parámetros de los algoritmos

Como se comentó en la subsección 3.1.1 cada algoritmo tiene sus propios parámetros que deben ser establecidos antes de realizar las simulaciones y se definieron con base a recomendaciones existentes en la literatura.

Para evolución diferencial (mono-objetivo y multi-objetivo):

- $S = [(0, 10), (0, 1), (0, 0.2)]$
- $NP = 30$

- $G = 50$
- $F = 0.5$
- $CR = 0.85$

Para optimización por enjambre de partículas:

- $S = [(0, 10), (0, 1), (0, 0.2)]$
- $NP = 30$
- $M = 85$
- $w = 0.2$
- $c_1 = 1$
- $c_2 = 2$

4.2. Optimización mono-objetivo

El objetivo de llevar a cabo este enfoque de optimización es observar con qué criterio (ver subsecciones 2.4.2.1 y 2.4.2.2) utilizado como función objetivo se genera un mejor sobreimpulso, tiempo de asentamiento y esfuerzo de control, así como, verificar qué metaheurística tiene un mejor desempeño con el mismo esfuerzo computacional y para el mismo criterio de optimización.

La sintonización del controlador se aplicó para un sistema lineal de segundo orden con retardo en la entrada. La respuesta en lazo abierto del sistema con un retardo de 0.2 segundos en la entrada a un escalón unitario se presenta en la Figura 11.

Los resultados que se presentan a continuación se compararon con las respuestas del sistema utilizando como parámetros del controlador PID los obtenidos con Ziegler-Nichols, `pidtune()` de MATLAB® y los resultados de KPSO (Chen *et al.*, 2017). Cabe mencionar que la sintonización llevada por los dos primeros métodos es mediante el uso de la función de transferencia.

Los valores del controlador PID y las principales características de la respuesta del sistema con retardo y de la señal de control obtenidos con cada uno de los métodos mencionados se muestran en la Tabla 3. En la primera columna se presentan los algoritmos utilizados para la sintonización del controlador, de la segunda columna a la cuarta están los parámetros del controlador PID obtenidos con dichos algoritmos

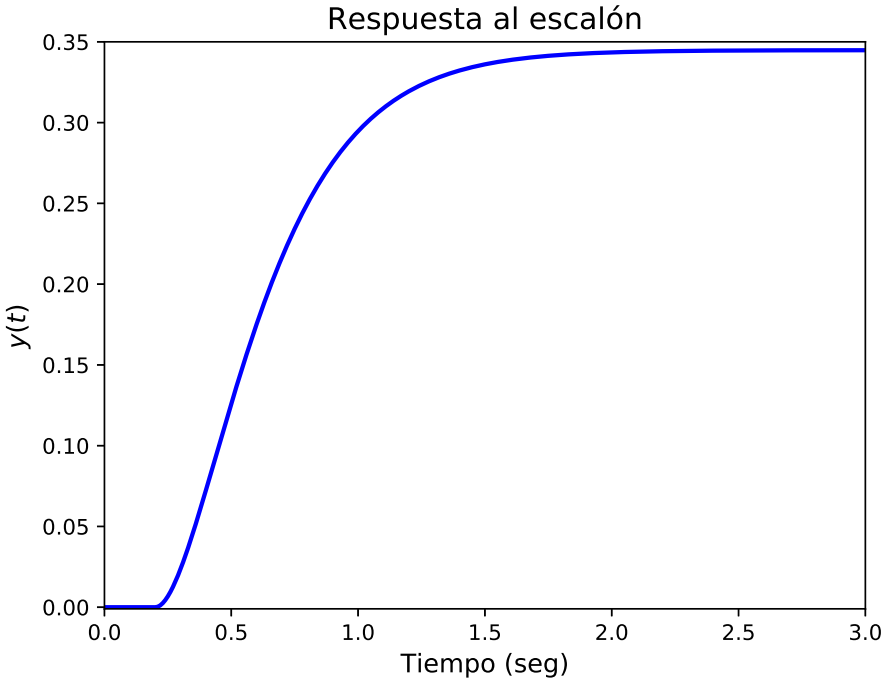

Figura 11. Respuesta del sistema en lazo abierto de segundo orden a una entrada escalón.

(K_p, T_i, T_d) y en las columnas restantes se presentan características del sistema de control con retardo, como el sobreimpulso (M_p), el tiempo de asentamiento (t_s), las normas 1, 2 e infinito de la ley de control ($||u||_1, ||u||_2, ||u||_\infty$).

Tabla 3. Parámetros del controlador PID y características principales.

Algoritmo	K_p	T_i	T_d	M_p (%)	t_s (seg)	$\|\|u\|\|_1$	$\|\|u\|\|_2$	$\|\|u\|\|_\infty$
`pidtune()`	4.11	0.525	0.117	17.91	1.16	130.60	6.53	5.67
Z-N	5.1939	0.5155	0.1289	30.15	1.66	145.26	7.26	7.20
KPSO	3.2140	0.4789	0.1195	11.19	1.53	120.97	6.04	4.55

4.2.1. Evolución diferencial

Utilizando los datos propuestos en la subsección 4.1.1, se realizó la sintonización usando diferentes funciones objetivo. Los parámetros obtenidos para cada uno de estas funciones objetivo se muestran en la Tabla 4. En la primer columna se presentan las funciones objetivo utilizadas para optimizar la sintonización del controlador, de la segunda columna a la cuarta están los parámetros del controlador PID obtenidos con dicha función objetivo (K_p, T_i, T_d) y en las columnas restantes se presentan características del sistema de control con retardo, como el tiempo de asentamiento (t_s), las normas 1, 2 e infinito de la ley de control ($||u||_1, ||u||_2, ||u||_\infty$) y por último el tiempo de ejecución del algoritmo (T_e).

Tabla 4. Parámetros del controlador PID y características principales de la respuesta del sistema con retardo obtenidos con el algoritmo evolución diferencial.

F. objetivo	K_p	T_i	T_d	$t_s\,(seg)$	$\|\|\mathscr{U}\|\|_1$	$\|\|\mathscr{U}\|\|_2$	$\|\|\mathscr{U}\|\|_\infty$	$T_e\,(seg)$
ITAE	3.790192	0.650032	0.147204	0.600	120.71	6.04	4.96	6.62
IAE	4.632969	0.773373	0.167862	1.146	137.89	6.89	5.83	6.55
IE	4.585318	0.765040	0.166816	1.147	137.03	6.85	5.78	6.42
ISE	5.228465	0.921154	0.181989	1.761	157.38	7.87	6.36	6.88
ITSE	4.901010	0.828807	0.174142	1.131	145.53	7.28	6.08	6.74
ISTAE	3.716135	0.641025	0.146053	0.610	119.47	5.97	4.88	6.72
ISTSE	4.484506	0.747056	0.164158	1.149	134.16	6.71	5.69	6.64
M_p	1.902187	0.551269	0.175392	2.267	98.67	4.93	2.90	7.27
t_s	3.814844	0.652826	0.148165	0.590	121.03	6.05	4.98	6.65

Con todas las funciones objetivo se obtiene un sobreimpulso prácticamente de cero por ciento. En la tabla se puede observar que el menor tiempo de asentamiento se obtiene cuando se utiliza como criterio el tiempo de asentamiento, seguido de los índices ITAE e ISTAE con un valor de 0.6 segundos y 0.61 segundos, respectivamente. Respecto al esfuerzo de control se puede apreciar que cuando se utiliza el sobreimpulso como criterio de optimización se obtiene el menor esfuerzo posible, teniendo un valor de 4.60 unidades, pero conlleva un tiempo de asentamiento cuatro veces más tardado que los logrados con los criterios ITAE, ISTAE y t_s. En términos generales la optimización del índice ISTAE es la que presenta mejores características en la respuesta del sistema.

Los valores que se muestran en la Tabla 4, son los mejores parámetros de treinta simulaciones realizadas, tomando en cuenta el menor sobreimpulso y el menor tiempo de asentamiento. Los datos estadísticos de los resultados para la optimización de los criterios como función objetivo y del tiempo de asentamiento de las respuestas del sistema se muestran en las tablas 5 y 6. Los datos estadísticos del sobreimpulso se omitieron debido a que resultaban en un valor de cero en todas las funciones objetivo.

La evolución de cada una de las funciones de costo conforme van pasando las generaciones se muestra en la Figura 12. En ella se puede apreciar que después de un cierto número de generaciones transcurridas la función de costo converge a un valor finito, el cual podemos declarar como un mínimo local, a excepción de cuando se utiliza el sobreimpulso como función objetivo, ya que, desde el inicio se queda en el mínimo. Los valores en las gráficas están normalizados de cero a uno con excepción del tiempo de asentamiento.

Tabla 5. Estadísticas de la optimización de las funciones de costo con DE.

Función de costo	Media	Mediana	Desv. estándar
ITAE	0.6146	0.6142	0.0024
IAE	2.8836	2.8823	0.0038
IE	2.8837	2.8827	0.0033
ISE	1.7682	1.7674	0.0027
ITSE	0.2262	0.2262	0.0000
ISTAE	0.2293	0.2278	0.0040
ISTSE	0.0498	0.0498	0.0000
M_p	0.0000	0.0000	0.0000
t_s	0.8417	0.7626	0.1834

Tabla 6. Estadísticas del tiempo de asentamiento de las respuestas al escalón con DE.

Función de costo	Media	Mediana	Desv. estándar
ITAE	1.0979	1.1252	0.0957
IAE	1.1493	1.1492	0.0026
IE	1.1487	1.1486	0.0014
ISE	2.2870	2.3449	0.1760
ITSE	1.1311	1.1309	0.0003
ISTAE	0.6365	0.6303	0.0130
ISTSE	1.1515	1.1516	0.0008
M_p	16.0646	21.5029	9.6323
t_s	0.6125	0.6000	0.0336

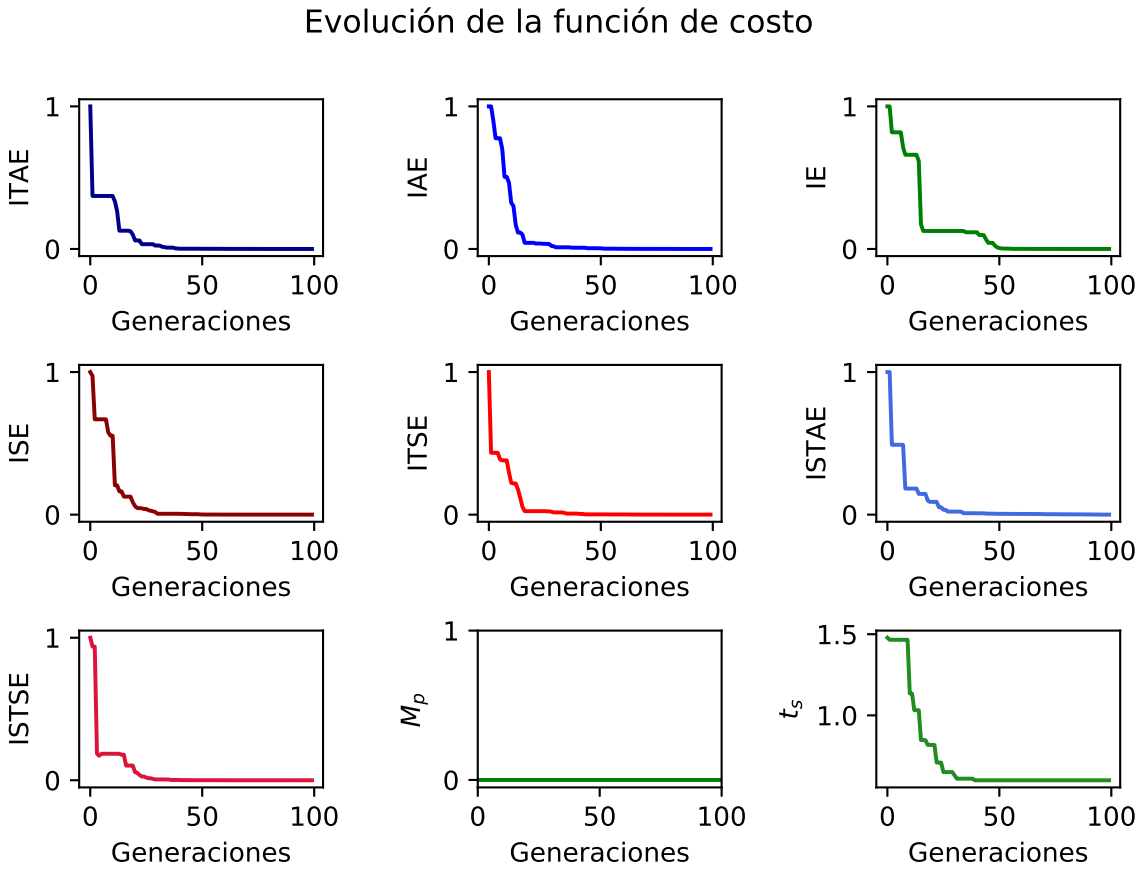

Figura 12. Evolución de las funciones de costo optimizadas con evolución diferencial.

Las respuestas del sistema ante una entrada escalón unitario, para cada uno de los criterios que se utilizaron para obtener los parámetros sintonizados se presentan en las figuras 13 y 14. De igual forma, se realiza la comparación gráfica con las técnicas presentadas en la sección 4.2.

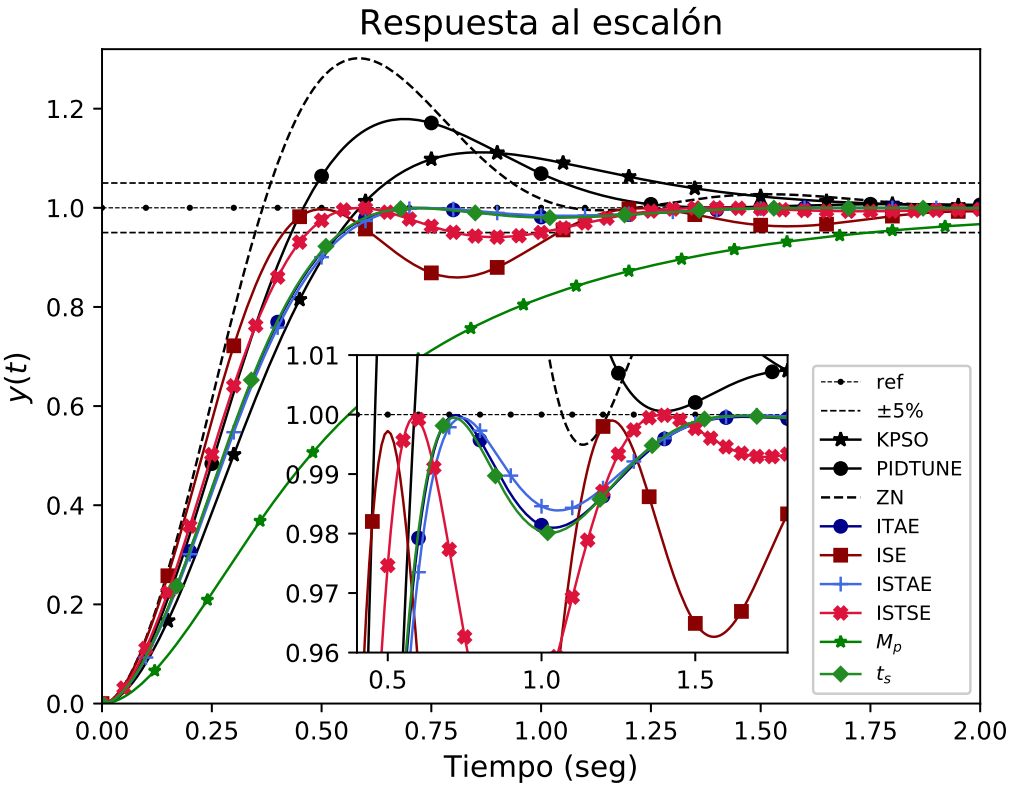

Figura 13. Respuestas del sistema de control a una entrada escalón optimizadas con evolución diferencial para los criterios de ITAE, ISE, ISTAE, ISTSE, M_p, t_s incluyendo los métodos KPSO, `pidtune()` y Z-N.

Las señales de control para cada una de las respuestas del sistema se presentan en las figuras 15 y 16. De estas gráficas, se puede inferir que a mayor tiempo de asentamiento el esfuerzo de control es menor, esto observando el caso cuando se utiliza el criterio de sobreimpulso (M_p). Sin embargo, también cuando existe un sobreimpulso elevado, como en el caso de la heurística de Z-N, el esfuerzo de control resulta tambíen en un valor alto. Lo recomendable en este caso es obtener una respuesta sin sobreimpulso, pero con un tiempo de asentamiento relativamente pequeño, para no obtener un esfuerzo de control mayor.

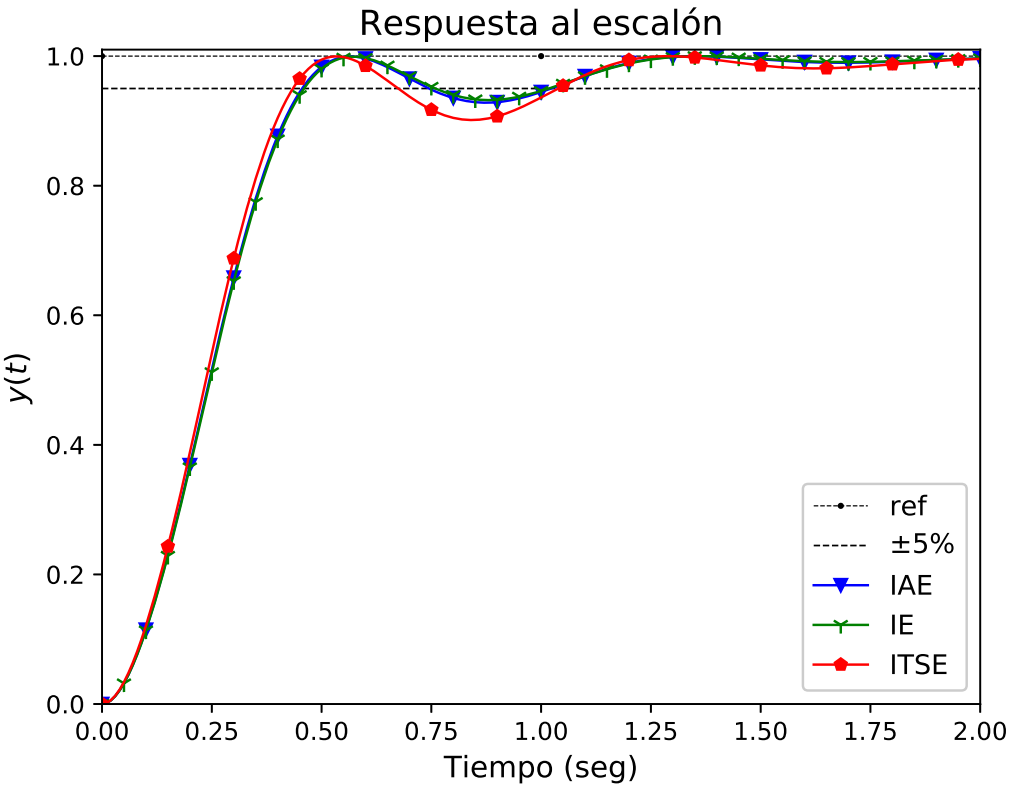

Figura 14. Respuestas del sistema de control a una entrada escalón optimizadas con evolución diferencial para los criterios de IAE, IE, ITSE y t_s.

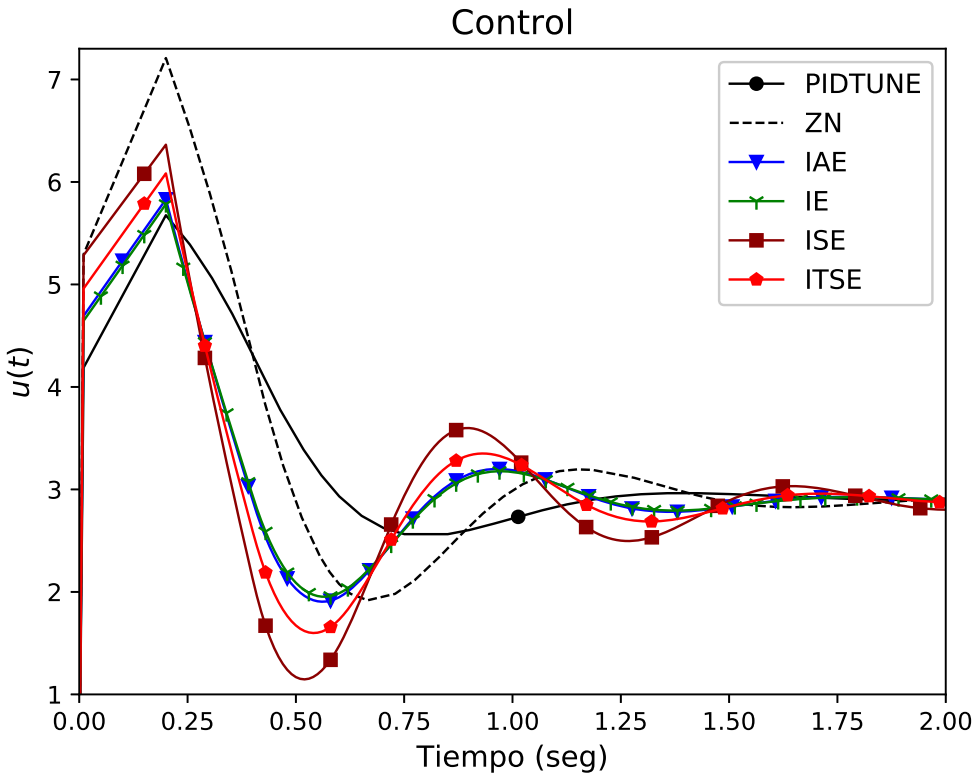

Figura 15. Señales del control PID sintonizado con evolución diferencial utilizando los criterios de optimización de IAE, IE, ISE e ITSE, incluyendo los métodos de `pidtune` y Z-N.

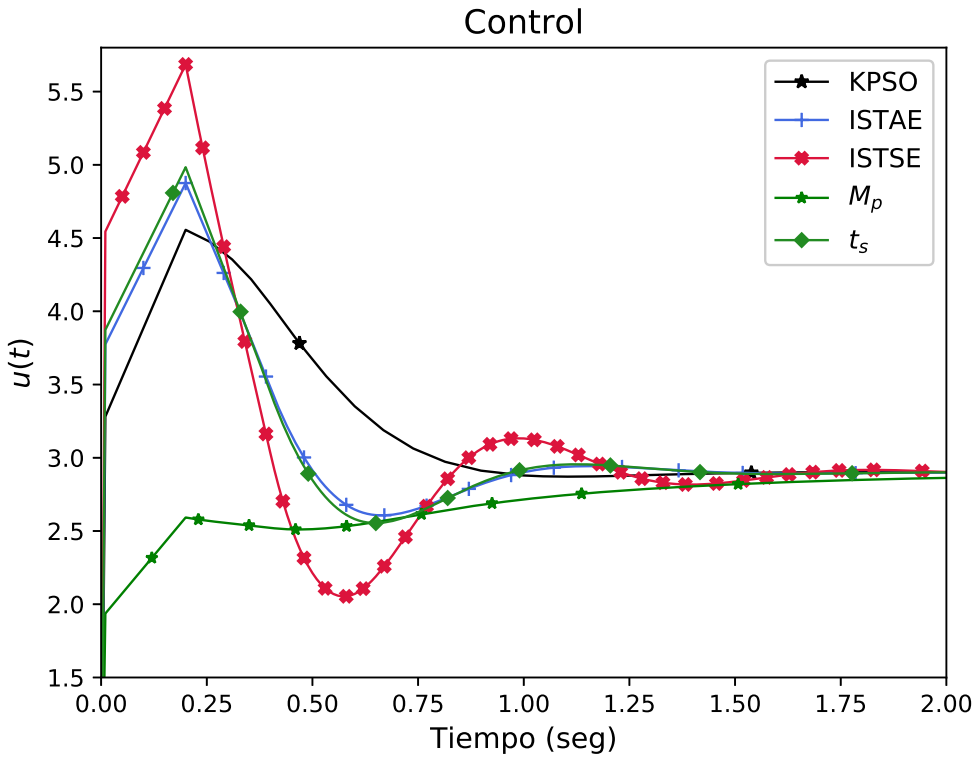

Figura 16. Señales del control PID sintonizado con evolución diferencial utilizando los criterios de optimización de ISTAE, ISTSE, M_p y t_s, incluyendo el método KPSO.

4.2.2. Optimización por enjambre de partículas

Minimizando con PSO los parámetros obtenidos para cada una de estas funciones objetivo se muestran en la Tabla 7. En la primera columna se presentan las funciones objetivo utilizadas para la sintonización del controlador, de la segunda columna a la cuarta están los parámetros del controlador PID obtenidos con dicha función objetivo (K_p, T_i, T_d) y en las columnas restantes se presentan características del sistema de control con retardo, como el tiempo de asentamiento (t_s), las normas 1, 2 e infinito de la ley de control ($||u||_1, ||u||_2, ||u||_\infty$) y por último el tiempo de ejecución del algoritmo (T_e). Para este algoritmo se utilizaron los parámetros expuestos en la subsección 3.1.1.

Con todos los criterios se obtiene un sobreimpulso de cero por ciento. En esta tabla se puede apreciar que respecto al tiempo de asentamiento los mejores resultados se obtienen al utilizar ISTAE e ISTSE con 0.61 segundos, seguidos del índice ITAE con 0.62 segundos. Tomando en cuenta el esfuerzo de control, el menor se obtiene al utilizar el criterio de sobreimpulso, pero tiene el inconveniente de que su tiempo de asentamiento es grande en comparación con los demás criterios.

Tabla 7. Parámetros del controlador PID y características principales de la respuesta del sistema con retardo obtenidos con el algoritmo optimización por enjambre de partículas.

F. objetivo	K_p	T_i	T_d	$t_s\,(seg)$	$\|\|u\|\|_1$	$\|\|u\|\|_2$	$\|\|u\|\|_\infty$	$T_e\,(seg)$
ITAE	3.702403	0.645994	0.144903	0.620	119.61	5.98	4.85	7.07
IAE	4.721776	0.810788	0.181008	1.121	142.69	7.13	5.89	6.64
IE	3.387856	0.620891	0.133784	0.696	116.62	5.83	4.48	6.62
ISE	4.865488	0.856992	0.175484	1.740	144.98	7.25	6.00	7.39
ITSE	4.586593	0.774983	0.172776	1.135	137.71	6.89	5.77	6.51
ISTAE	3.734435	0.647677	0.145706	0.610	119.98	6.00	4.89	6.90
ISTSE	3.720283	0.642397	0.145357	0.610	119.75	5.99	4.88	6.78
M_p	3.230331	0.598026	0.130702	0.721	114.42	5.72	4.31	7.739
t_s	3.667410	0.642644	0.147973	0.640	118.94	5.95	4.81	7.08

Los datos estadísticos de los resultados para la optimización de la sintonización del controlador PID de treinta simulaciones se presentan en las tablas 8 y 9.

Tabla 8. Estadísticas de la optimización de las funciones de costo con PSO.

Función de costo	**Media**	**Mediana**	**Desv. estándar**
ITAE	0.7403	0.7065	0.1034
IAE	3.0036	3.0244	0.2552
IE	2.0533	2.2091	1.1130
ISE	1.2897	1.3883	0.5246
ITSE	0.2403	0.2352	0.0165
ISTAE	0.3913	0.3597	0.1773
ISTSE	0.0565	0.0559	0.0039
$\boldsymbol{M_p}$	0.2902	0.0000	0.3898
$\boldsymbol{t_s}$	1.0028	1.0445	0.2666

La evolución de cada una de las funciones de costo conforme van pasando las generaciones se presenta en la Figura 17. Al igual que en el caso anterior, en ellas se puede apreciar que después de un cierto número de generaciones transcurridas la función de costo converge a un determinado valor, el cual podemos declarar como un mínimo local, a excepción de cuando se utiliza el sobreimpulso como función objetivo, ya que, de nuevo se obtiene el mismo comportamiento de quedarse en el mínimo desde el inicio. Los costos en las gráficas están normalizados de cero a uno con excepción del tiempo de asentamiento.

Las respuestas del sistema ante una entrada escalón unitario, para cada uno de los criterios optimizados se presentan en las figuras 18 y 19 y las señales de control para cada una de las respuestas del sistema se presentan en las figuras 20 y 21.

Tabla 9. Estadísticas del sobreimpulso y tiempo de asentamiento de las respuestas al escalón con PSO.

Parámetro	Función de costo	Media	Mediana	Desv. estándar
M_p (%)	ITAE	0.0000	0.0000	0.0000
	IAE	0.2848	0.0000	1.5338
	IE	23.2197	5.5814	31.4219
	ISE	2.8429	0.0000	13.9323
	ITSE	0.0323	0.0000	0.1742
	ISTAE	0.0000	0.0000	0.0000
	ISTSE	0.0000	0.0000	0.0000
	M_p	0.2902	0.0000	0.3898
	t_s	0.0354	0.0000	0.1908
t_s (seg)	ITAE	1.1623	1.1575	0.1751
	IAE	3.0871	1.2230	5.5149
	IE	6.8678	1.8360	8.7691
	ISE	12.6499	14.0855	9.971
	ITSE	1.5479	1.6910	0.2754
	ISTAE	0.9416	0.8605	0.2177
	ISTSE	1.1788	1.1645	0.1494
	M_p	2.8075	1.2070	4.4706
	t_s	1.0029	1.0445	0.2665

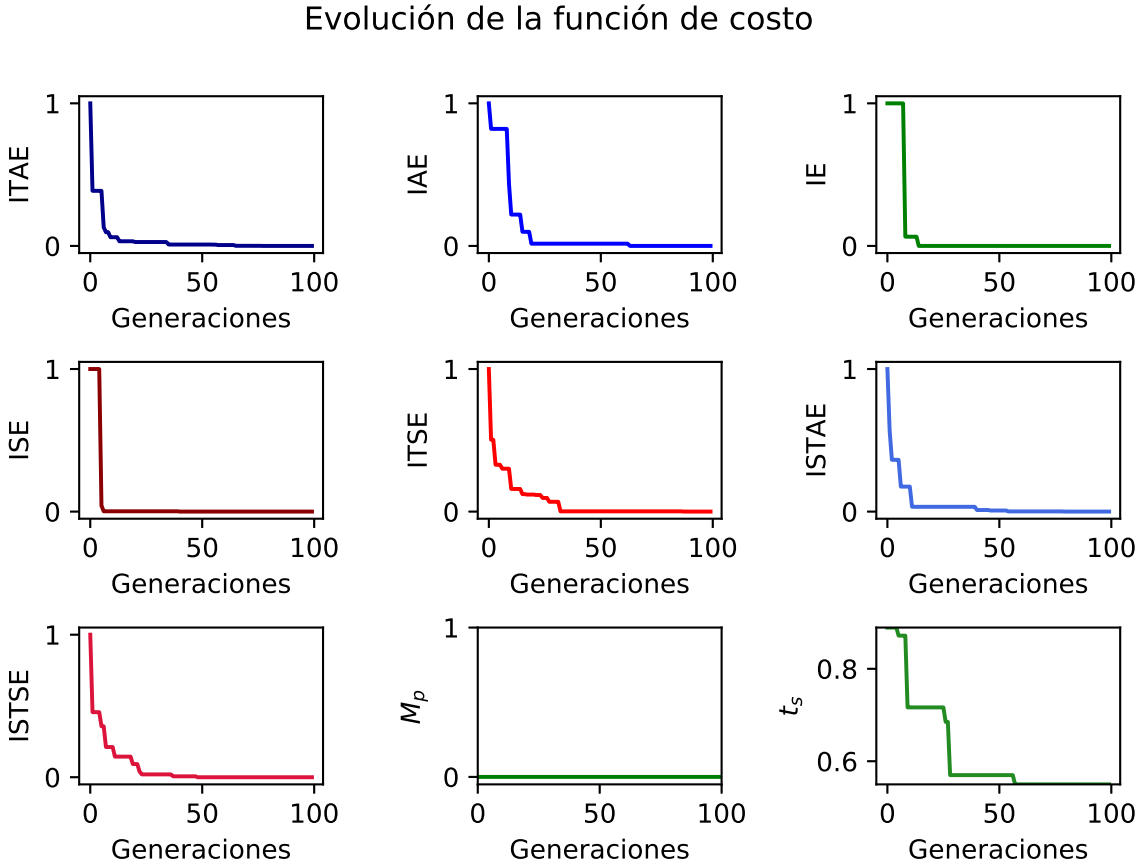

Figura 17. Evolución de las funciones de costo optimizadas con optimización por enjambre de partículas.

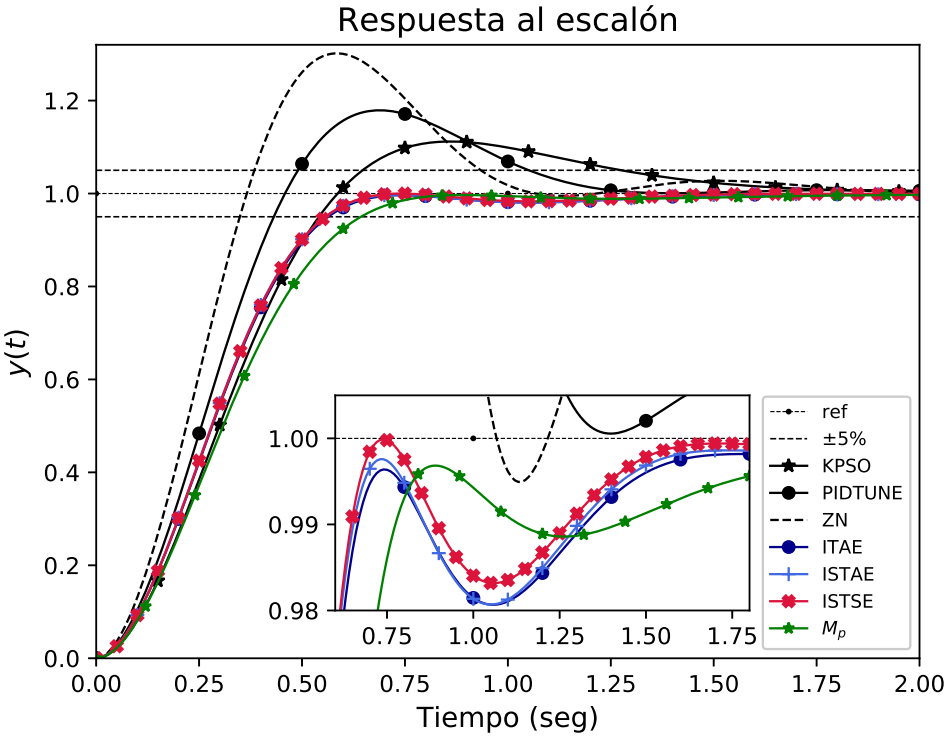

Figura 18. Respuestas del sistema de control a una entrada escalón optimizada con optimización por enjambre de partículas para los criterios de ITAE, ISTAE, ISTSE y M_p, incluyendo los métodos KPSO, `pidtune()` y Z-N.

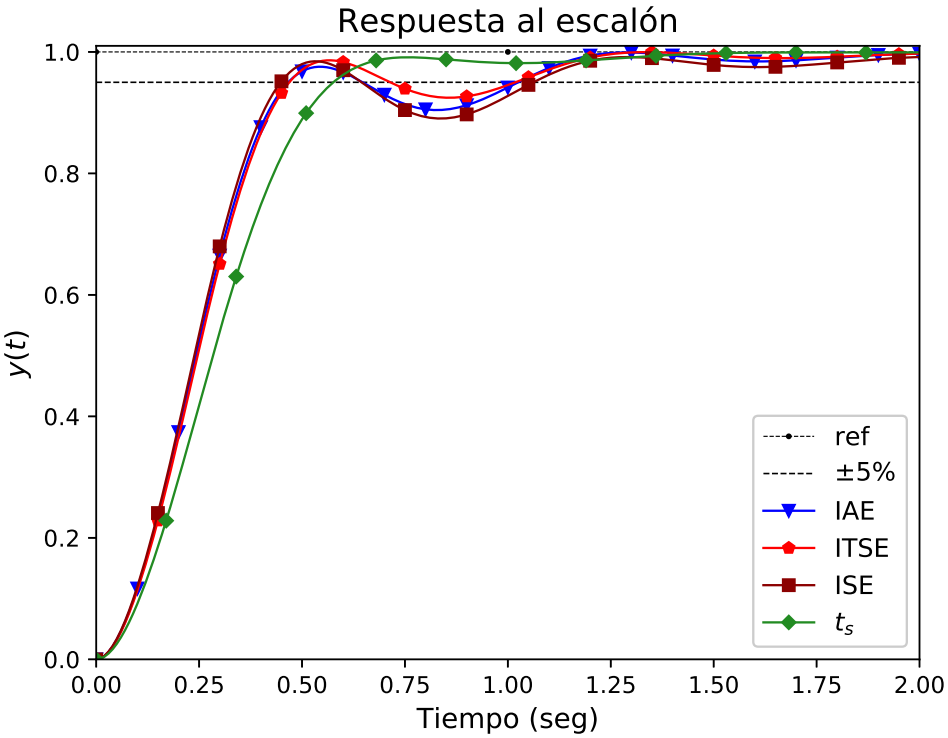

Figura 19. Respuestas del sistema de control a una entrada escalón optimizada con optimización por enjambre de partículas para los criterios de IAE, ITSE, ISE y t_s.

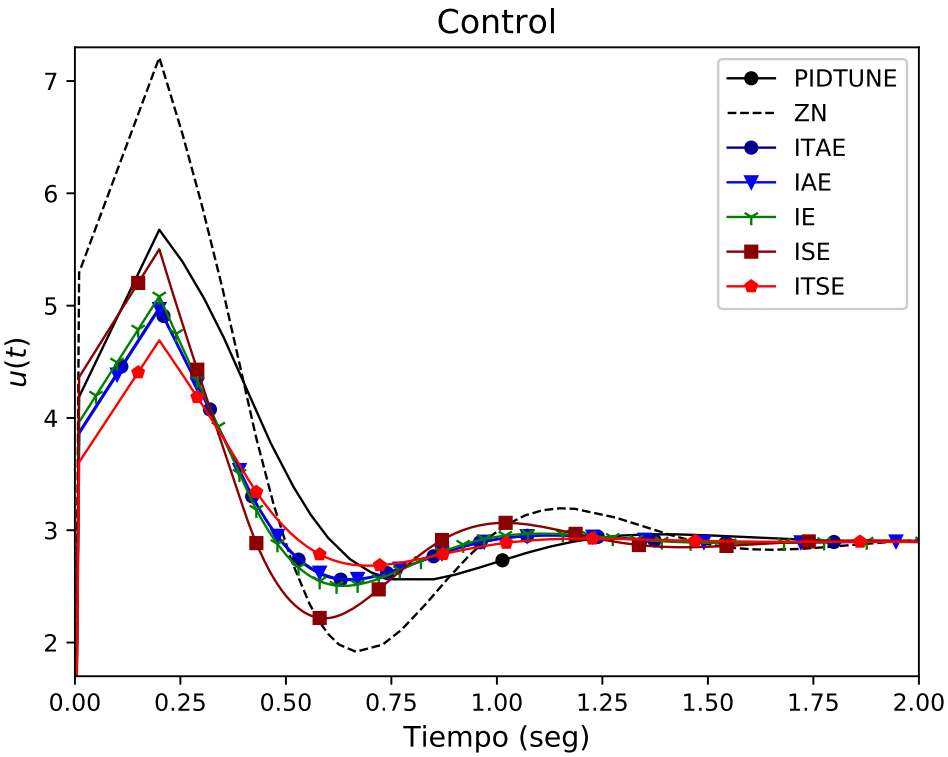

Figura 20. Señales del control PID sintonizado con optimización por enjambre de partículas utilizando los criterios de optimización de ITAE, IAE, IE, ISE e ITSE, incluyendo los métodos de `pidtune` y Z-N.

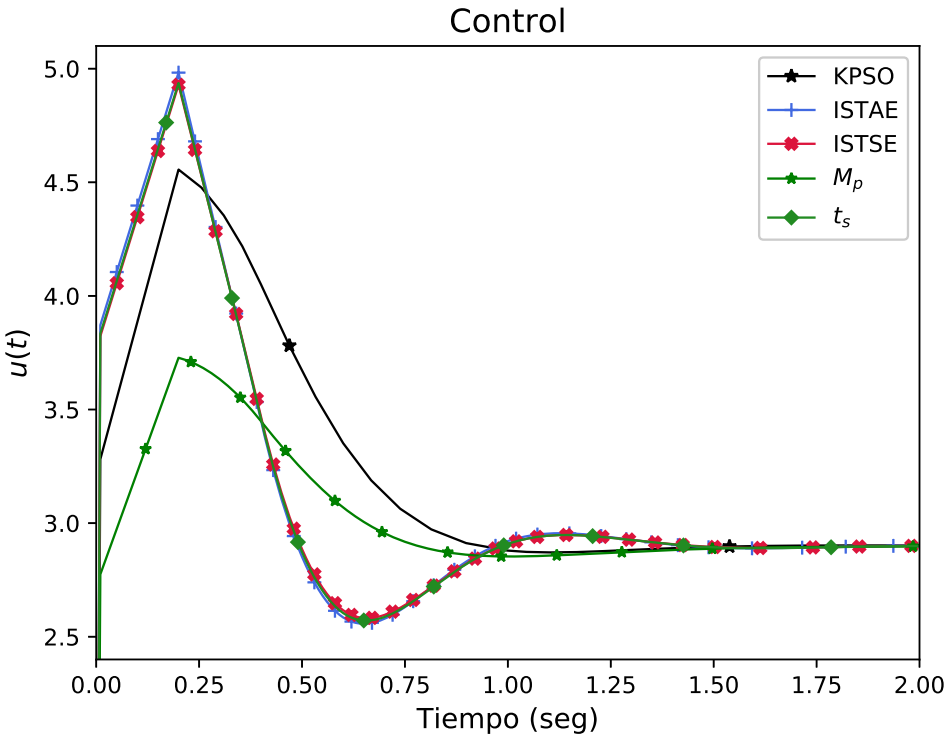

Figura 21. Señales del control PID sintonizado con optimización por enjambre de partículas utilizando los criterios de optimización de ISTAE, ISTSE, M_p y t_s, incluyendo el método KPSO.

4.3. Comparativa de desempeño de los algoritmos

Para determinar cuál algoritmo de optimización presentó un mejor desempeño al sintonizar el controlador PID, se optó por utilizar las gráficas de radar. La primer comparación que se realizó fue utilizando el mejor resultado obtenido por cada algoritmo para cada una de las funciones objetivo. Cabe mencionar que todos los valores presentes en las figuras subsecuentes están normalizados a uno tomando en cuenta el máximo valor presentado entre los dos algoritmos.

El algoritmo evolución diferencial presenta mejor resultado en los índices IE, IAE e ISE esto se puede apreciar en la Figura 22. Realizando una magnificación en la parte inferior de la gráfica (ver Figura 23), se puede determinar que en el resto de los criterios la metaheurística DE también presenta un mejor resultado que PSO.

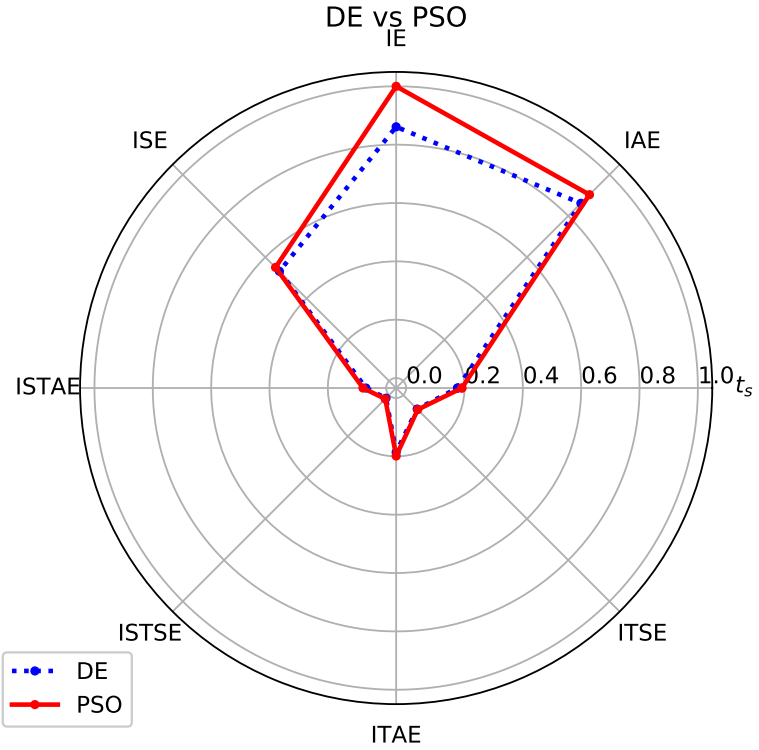

Figura 22. Comparación de mejor resultado entre DE y PSO.

Así mismo, dentro de la comparativa de los algoritmos, se realizaron treinta simulaciones para cada una de las funciones objetivo. Para determinar que las diferencias entre las medias y las medianas de ambos algoritmos (ver tablas 5 y 8) fueran estadísticamente significativas, se realizó la prueba de Wilcoxon (Siegel, 1956), obteniendo un resultado de 99% de significancia. Esto indica que las diferencias

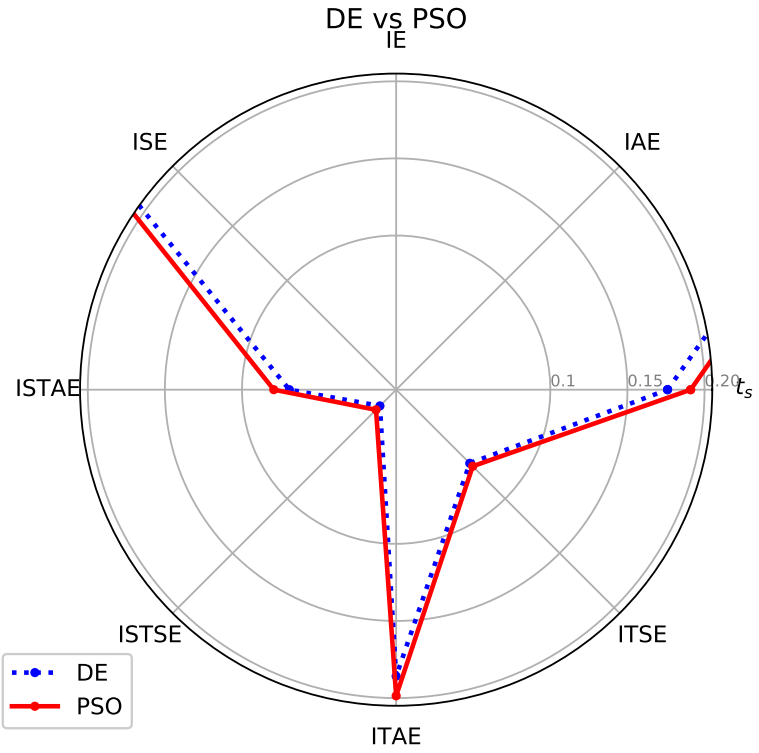

Figura 23. Acercamiento de comparación de mejor resultado entre DE y PSO.

observadas en la Figura 24 se deben a los procesos estocásticos subyacentes (los algoritmos) y no al azar.

Mediante una gráfica de radar se realizó la comparación entre evolución diferencial y optimización por enjambre de partículas (ver Figura 24). En ella se puede apreciar que el algoritmo de optimización por enjambre de partículas presenta mejores resultados para los índices IE e ISE, sin embargo, para el resto de funciones objetivo, el algoritmo evolución diferencial presenta mejores resultados, y se puede apreciar mejor al realizar un acercamiento en la parte inferior (ver Figura 25).

Con base a los resultados obtenidos y al análisis estadístico de los mismos, el algoritmo que mejores resultados presenta al utilizar optimización mono-objetivo es evolución diferencial. Aunque los resultados de las funciones objetivo, que se pueden ver en las tablas 4 y 7, no muestren diferencia significativa alguna, se obtiene que la dispersión con respecto a la media con evolución diferencial es considerablemente de menor grado que con optimización por enjambre de partículas en cada una de dichas funciones objetivo. Esta diferencia puede ocurrir por el hecho de utilizar un sobreimpulso (M_p) y un tiempo de asentamiento (t_s) restringidos, ya que cuando estos se omitían el grado de la dispersión disminuía.

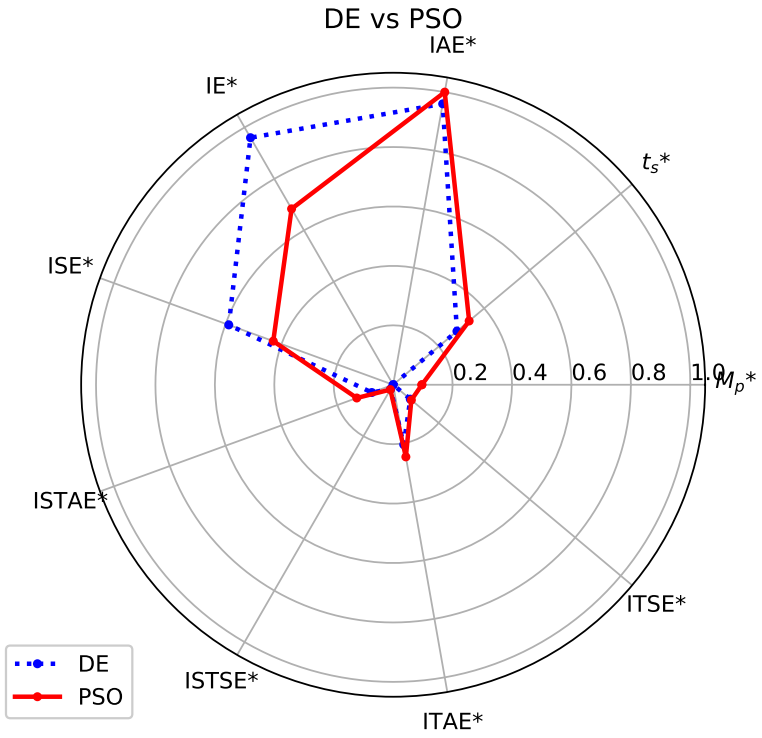

Figura 24. Comparación de resultado promedio entre DE y PSO.

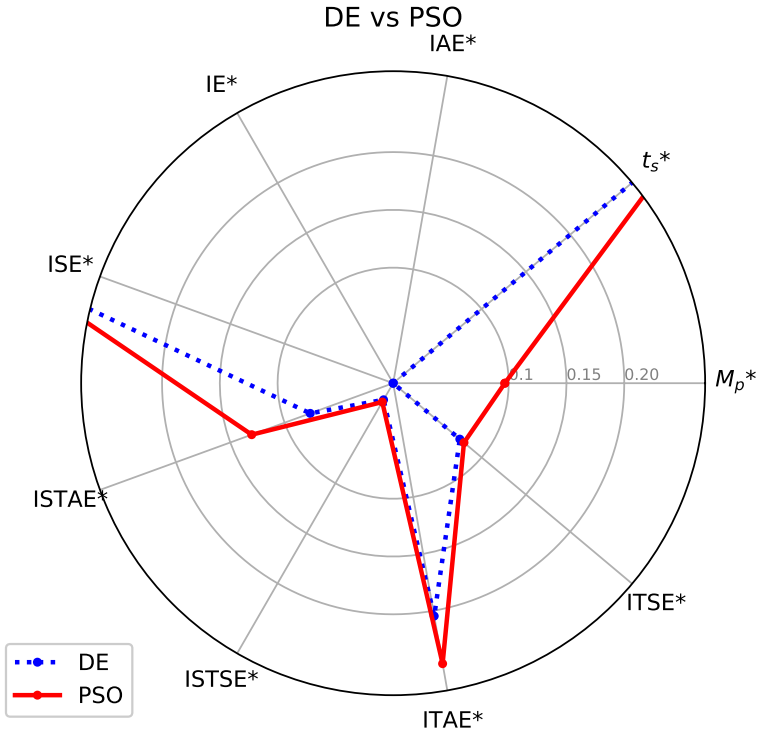

Figura 25. Acercamiento de comparación de resultado promedio entre DE y PSO.

Los valores de las funciones objetivo deben ser interpretados cuidadosamente ya que se tienen características muy diferentes. Con los resultados obtenidos por los algoritmos de optimización, la respuesta del sistema de control con retardo más adecuada que presenta un sombreimpulso nulo, tiene un tiempo de asentamiento pequeño y que además su esfuerzo de control no es grande, se obtiene al utilizar como función de costo, los índices de desempeño ITAE e ISTAE en ambos algoritmos, aunque puede considerarse el tiempo de asentamiento como otra opción, ya que presenta resultados favorables.

Además de los resultados de búsqueda, es necesario evaluar el desempeño de los algoritmos utilizados; en este caso, el costo computacional medido en tiempo de ejecución. Los tiempos de ejecución se pueden observar en la última columna de las tablas 4 y 7. En un inicio las simulaciones de optimización fueron ajustadas a cincuenta iteraciones con una población de treinta individuos para cada algoritmo, con lo cual se permitía realizar una comparación adecuada. Realizando esto, se pudo apreciar que el algoritmo de optimización por enjambre de partículas realizaba sus búsquedas en la mitad de tiempo que el algoritmo evolución diferencial, por lo que se decidió elevar el número de iteraciones al algoritmo PSO, para igualar el costo computacional y observar si se obtenían mejores resultados.

4.4. Optimización multi-objetivo

Para analizar la relación de compromiso entre los criterios de optimización, se optó por realizar una optimización multi-objetivo con la metaheurística DEMO. Para esto se utilizó como función objetivo, el sobreimpulso (M_p), el tiempo de asentamiento (t_s) y el esfuerzo de control ($||u||_\infty$). Todos estos criterios fueron optimizados simultáneamente. Se realizaron dos simulaciones, una utilizando como funciones objetivo el sobreimpulso (M_p) y tiempo de asentamiento (t_s), y una segunda usando tres funciones objetivos: las dos ya mencionadas y la norma infinito de la ley de control ($||u||_\infty$).

4.4.1. Optimización con dos funciones objetivo

En esta simulación se utilizaron como funciones objetivo el sobreimpulso (M_p) y el tiempo de asentamiento (t_s). Para tener datos estadísticamente significativos se

realizó una muestra de treinta simulaciones, donde en cada una se obtenía el frente no dominado del conjunto de soluciones. De todos los resultados obtenidos de estas simulaciones se consolidó un aproximado de Pareto.

A diferencia de la optimización mono-objetivo en la cual sólo se obtenía un único resultado, aquí se tiene un conjunto de soluciones, que conforme a los requerimientos de la respuesta, se puede seleccionar cualquiera de ellos.

El frente consolidado no dominado que conforman las mejores soluciones para la sintonización del controlador PID se puede apreciar en la Figura 26. A simple vista se infiere que `pidtune()` y KPSO son fuertemente dominados por el conjunto de soluciones. Además se muestra la respuesta del sistema de control sintonizado con algunas de las soluciones. Si las comparamos con las respuestas del sistema de control sintonizado con `pidtune()` y KPSO, se ve a detalle que es superior a ambas en las características de sobreimpulso y tiempo de asentamiento. También se puede ver que existe un conflicto entre el sobreimpulso y tiempo de asentamiento, ya que al mejorar uno, el otro empeora.

Cinco de los once conjuntos de parámetros obtenidos en el frente no dominado se presentan en la Tabla 10, de los cuales los conjuntos 1 y 2 son las soluciones de los extremos, 3 y 4 son soluciones intermedias y 5 es la solución que se encuentra en el punto medio del conjunto. Se puede ver que uno de los extremos es el que mejor relación presenta entre las dos funciones objetivo, ya que no existe sobreimpulso y el tiempo de asentamiento es similar al de los demás (de hecho la diferencia es de solo 0.1 segundos). Si se observa detalladamente entre las soluciones uno y dos, cuando existe un tiempo de asentamiento menor, el esfuerzo de control aumenta. Esto está en congruencia con los resultados obtenidos con optimización mono-objetivo, en donde con los distintos criterios se encontró que el que presentaba menor tiempo de asentamiento, era el que más esfuerzo de control requería.

Con este conjunto de mejores soluciones o frente de Pareto, un especialista en diseño de sistemas de control, puede seleccionar la que mejor se adapte a sus requerimientos, e independientemente de cuál se seleccione, al final de cuentas todas son soluciones válidas con las que se obtienen las mejores respuestas del sistema. Además se tiene un mejor resultado que con la herramienta de MATLAB®, la cual

Tabla 10. Tabla con los parámetros del controlador PID y características principales de la respuesta del sistema optimizando con dos funciones objetivo.

Solución	K_p	T_i	T_d	M_p (%)	t_s (seg)	$\|\|u\|\|_1$	$\|\|u\|\|_2$	$\|\|u\|\|_\infty$
1	3.774827	0.649554	0.147461	0.0	0.6	120.37	6.01	4.94
2	4.284375	0.663259	0.164760	1.8	0.5	130.75	6.53	5.58
3	3.913140	0.657240	0.151992	0.23	0.57	123.02	6.15	5.1
4	4.099841	0.659994	0.158764	0.97	0.53	126.48	6.32	5.34
5	3.999322	0.658672	0.154738	0.59	0.55	124.73	6.23	5.21

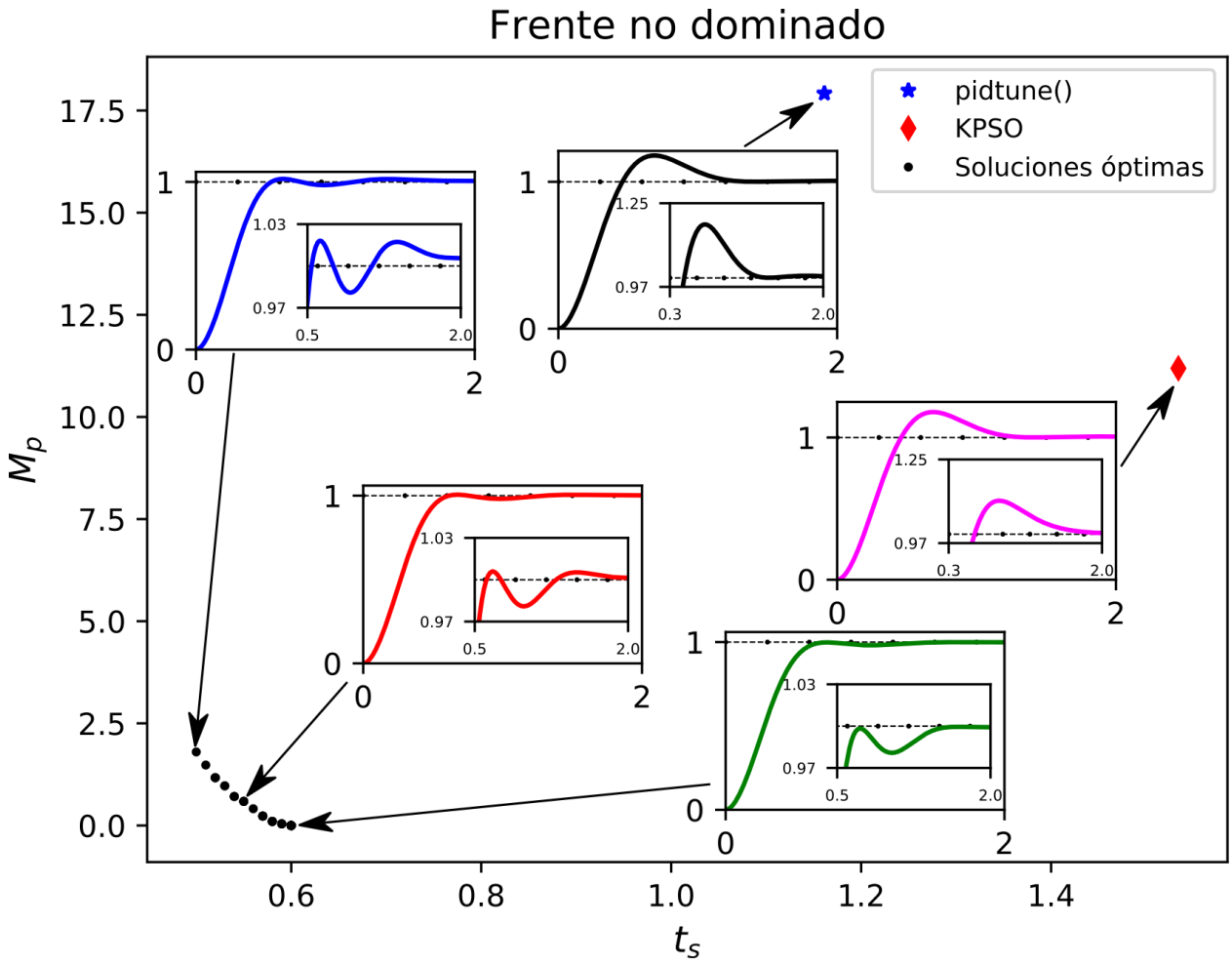

Figura 26. Frente no dominado de soluciones óptimas obtenido con la metaheurística DEMO.

se utilizó con su comando por omisión. En este caso se desconoce sí utilizando las opciones que tiene dicha herramienta, se logre alcanzar o superar el resultado obtenido con el algoritmo DEMO.

4.4.2. Optimización con tres funciones objetivo

En este caso se usaron como funciones objetivo el sobreimpulso, el tiempo de asentamiento y el esfuerzo de control. De igual forma al caso anterior, se propusieron estas funciones de costo para comprobar si existía conflicto entre ellas. Aunado a esto, como se observó que `pidtune` presentaba resultados inferiores en sobreimpulso y tiempo de asentamiento, se quería conocer si en esfuerzo de control también quedaba por debajo de los resultados obtenidos con DEMO. También se realizaron treinta simulaciones y se obtuvo en cada una de ellas el frente no dominado, consolidando al final un aproximado de Pareto con base a estas soluciones.

El frente consolidado no dominado con las soluciones obtenidas con `pidtune()` y KPSO se puede observar en la Figura 27. En las proyecciones de las soluciones se puede apreciar que el aproximado de Pareto (puntos en gris) es mejor en su mayoría a la solución obtenida con `pidtune()` (puntos en azul) y a la solución de KPSO (puntos en rosa), a excepción de pocas soluciones, que son peores en tiempo de asentamiento (t_s).

El optimizar con tres funciones objetivo ayuda a verificar que el esfuerzo de control ($||u||_\infty$) está en conflicto directamente con el tiempo de asentamiento (t_s), ya que entre mayor sea el tiempo de asentamiento es menor el esfuerzo de control requerido.

Analizando los resultados obtenidos se observó que existe un conflicto entre el tiempo de asentamiento y el esfuerzo de control, esto se puede apreciar desde que se abordó el problema de la sintonización del controlador PID como un problema de optimización mono-objetivo. De esto también se obtiene que el sobreimpulso no tiene un conflicto evidente con el esfuerzo de control.

Con la optimización multi-objetivo se encontró un aproximado de Pareto con soluciones que optimizan las funciones objetivo utilizadas, sobreimpulso, tiempo de asentamiento y esfuerzo de control. De estas soluciones, según el requerimiento del controlador se puede utilizar cualquiera y se tendrá un desempeño óptimo. Comparado con la solución de la herramienta `pidtune()` y la de KPSO, el conjunto de soluciones que se obtuvo domina en su totalidad a dichas soluciones, esto con dos funciones objetivo. Al utilizar tres funciones objetivo se obtiene un aproximado de Pareto con

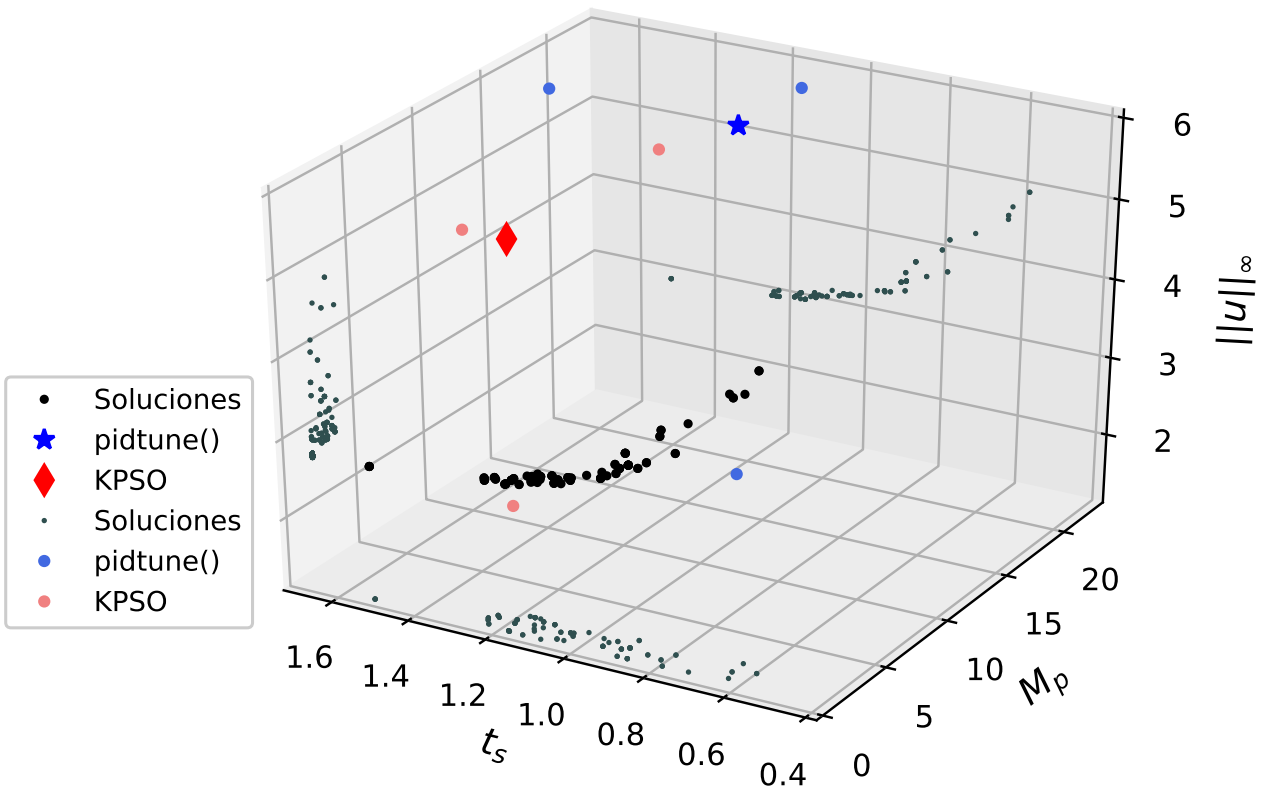

Figura 27. Frente no dominado de soluciones óptimas obtenido con la metaheurística DEMO.

mayor cantidad de soluciones de las que casi en su totalidad son mejores a la solución de `pidtune()` y la de KPSO.

Capítulo 5 Conclusiones

En esta tesis se propuso un procedimiento para la sintonización de un controlador PID para un sistema lineal de segundo orden con retardo en la entrada. En dicho procedimiento se considera el problema de sintonización como uno de optimización tanto mono- como multi-objetivo. La optimización se realizó con tres metaheurísticas: dos de evolución diferencial (versiones mono- y multi-objetivo) y una de optimización por enjambre de partículas. Las funciones objetivo consideradas fueron los índices de desempeño, el sobreimpulso, el tiempo de asentamiento y el esfuerzo de control. A continuación se presentan las conclusiones más importantes con base a los resultados obtenidos.

- Establecer como un problema de optimización mono-objetivo y multi-objetivo la sintonización de los parámetros del controlador PID para sistemas lineales de segundo orden con retardo resulta en una mejor sintonía en comparación a otras técnicas.
- La optimización de los índices de desempeño, ITAE, ISTAE, y el tiempo de asentamiento (t_s), lograron el mejor resultado en tiempo de asentamiento, así como un esfuerzo de control menor en comparación con los resultados obtenidos al optimizar los demás criterios. El criterio sobreimpulso es el que mejor resultado presentó en esfuerzo de control, pero su tiempo de asentamiento fue elevado.
- Con la metaheurística evolución diferencial se obtuvieron mejores resultados en las características particulares de la respuesta del sistema que utilizando la metaheurística de optimización por enjambre de partículas.
- Con optimización multi-objetivo se observó un grado de conflicto entre el sobreimpulso y el tiempo de asentamiento .
- Se observó que el esfuerzo de control está en conflicto en mayor grado con el tiempo de asentamiento que con el sobreimpulso.

La sintonización del controlador PID se llevó a cabo mediante simulaciones, y permitió obtener las siguientes inferencias:

- Estudios preliminares muestran que utilizar la heurística de Z-N para inicializar las soluciones de las metaheurísticas ayuda a reducir el costo computacional del algoritmo de búsqueda.

- Es recomedable utilizar como tiempo de simulación al menos diez veces el tiempo de asentamiento de la respuesta del sistema (en lazo abierto) a una entrada escalón unitaria, para tener mayor diversidad en las soluciones.

5.1. Trabajo a futuro

Como trabajo futuro se propone lo siguiente:

- Considerar la aplicación de la sintonización del controlador PID con sistemas no lineales con retardo.
- Usar restricciones de robustez en el problema de optimización.
- Utilizar las mismas metaheurísticas para sintonizar otros controladores.
- Comparar el desempeño de distintos controladores sintonizando los parámetros con la metodología propuesta.

Literatura citada

A. Rovira, A., W. Murrill, P., y L. Smith, C. (1970). Tuning controllers for set point changes. **42**: 25.

Åström, K. J. y Hägglund, T. (1995). *PID controllers: theory, design and tuning*. Instrument Society of America. p. 343.

Åström, K. J. y Hägglund, T. (2001). The future of PID control. *Control Engineering Practice*, **9**(11): 1163–1175.

Åström, K. J. y Hägglund, T. (2004). Revisiting the Ziegler-Nichols step response method for PID control. *Journal of Process Control*, **14**(6): 635–650.

Åström, K. J. y Hägglund, T. (2006). *Advanced PID control*. The Instrumentation, Systems, and Automation Society.

Bennett, S. (2001). The past of PID controllers. *Annual Reviews in Control*, **25**(4): 43–53.

Chen, J., Omidvar, M. N., Azad, M., y Yao, X. (2017). Knowledge-based particle swarm optimization for pid controller tuning. En: *2017 IEEE Congress on Evolutionary Computation (CEC)*, June. pp. 1819–1826.

Chien, K. L., Hrons, J. A., y Reswick, J. B. (1952). On the automatic control of generalized passive systems. *Transactions of the American Society of Mechanical Engineeing*, **74**: 175–185.

Chong, E. K. y Zak, S. H. (2013). *An introduction to optimization*, Vol. 76. John Wiley & Sons.

Cohen, G. y Coon, G. (1953). Theoretical consideration of retarded control. *Trans. ASME*, **75**: 827–834.

Deb, K., Pratap, A., Agarwal, S., y Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, **6**(2): 182–197.

Di Caro, G. y Dorigo, M. (1999). Ant colony optimization: a new meta-heuristic. En: *Proceedings of the 1999 Congress on Evolutionary Computation*. Vol. 2.

Dorf, R. C. y Bishop, R. H. (2011). *Modern control systems*. Pearson.

Flunkert, V. y Schoell, E. (2009). Pydelay-a python tool for solving delay differential equations. *arXiv preprint arXiv:0911.1633*.

Graham, D. y Lathrop, R. C. (1953). The synthesis of “optimum” transient response: criteria and standard forms. *Transactions of the American Institute of Electrical Engineers, Part II: Applications and Industry*, **72**(5): 273–288.

Gu, K., Chen, J., y Kharitonov, V. L. (2003). *Stability of time-delay systems*. Springer Science & Business Media.

Hägglund, T. y Åström, K. J. (2002). Revisiting the Ziegler-Nichols tuning rules for PI control. *Asian Journal of Control*, **4**(4): 364–380.

Hast, M., Åström, K. J., Bernhardsson, B., y Boyd, S. (2013). PID design by convex-concave optimization. *European Control Conference*, pp. 4460–4465.

He, J.-B., Wang, Q.-G., y Lee, T.-H. (2000). PI/PID controller tuning via LQR approach. *Chemical Engineering Science*, **55**(13): 2429–2439.

Ho, W. K., Hang, C. C., y Cao, L. S. (1995). Tuning of PID controllers based on gain and phase margin specifications. *Automatica*, **31**(3): 497–502.

Hu, W., Xiao, G., y Li, X. (2011). An analytical method for PID controller tuning with specified gain and phase margins for integral plus time delay processes. *ISA Transactions*, **50**(2): 268–276.

Kennedy, J. y Eberhart, R. (1995). Particle swarm optimization. En: *Proceedings of ICNN’95 - International Conference on Neural Networks*, Nov. Vol. 4, pp. 1942–1948 vol.4.

Kirkpatrick, S., Gelatt, C. D., y Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, **220**(4598): 671–680.

Kolmanovskii, V. y Myshkis, A. (1992). Applied theory of functional differential equations. 1992. *Kluwer Academic Publishers, Dordrecht)*, **4**: 400–408.

Liang, Y.-C. y Smith, A. (2004). An ant colony optimization algorithm for the redundancy allocation problem (RAP). *IEEE Transactions on Reliability*, **53**(3): 417–423.

Lopez, A., Murrill, P., y Smith, C. (1967). Controller tuning relationships based on integral performance criteria. *Instrumentation Technology*, **14**(11): 57.

Majumdar, S., Mandal, K. K., y Chakraborty, N. (2014). Tuning of robust PID controller using mine blast algorithm. En: *Proceedings of The 2014 International Conference on Control, Instrumentation, Energy and Communication (CIEC)*, jan. IEEE, pp. 752–756.

Martins, F. G. (2005). Tuning PID Controllers using the ITAE Criterion. *International Journal of Engineering Education*, **21**(5): 867–873.

Michiels, W. y Niculescu, S.-I. (2007). *Stability and Stabilization of Time-Delay Systems*. Society for Industrial and Applied Mathematics. Philadelphia, PA, USA.

Minorsky, N. (1922). Directional stability of automatically steered bodies. *Journal of ASNE*, **42**(2): 280–309.

O'Dwyer, A. (2009). *Handbook of PI and PID Controller Tuning Rules*. Imperial College Press, tercera edición. p. 623.

Ogata, K. (1990). *Modern Control Engineering*. Prentice Hall PTR, segunda edición. Upper Saddle River, NJ, USA.

Oliphant, T. E. (2007). Python for scientific computing. *Computing in Science & Engineering*, **9**(3).

Ouaarab, A., Ahiod, B., y Yang, X.-S. (2014). Discrete cuckoo search algorithm for the travelling salesman problem. *Neural Computing and Applications*, **24**(7-8): 1659–1669.

Pardalos, P. M., Žilinskas, A., y Žilinskas, J. (2017). *Non-convex multi-objective optimization*. Springer International Publishing.

Pareto, V. (1906). *Manuale di economia politica*, Vol. 13. Societa Editrice.

Pérez, F. y Granger, B. (2007). Python in scientific computing. *Computing in Science & Engineering*, **9**(3): 10–20.

Persson, P. (1992). *Towards Autonomous PID Control*. Tesis de doctorado, Department of Automatic Control.

Rao, A. R. (1993). *Process control engineering*. CRC Press.

Rivera, D. E., Morari, M., y Skogestad, S. (1986). Internal model control: Pid controller design. *Industrial & Engineering Chemistry Process Design and Development*, **25**(1): 252–265.

Robič, T. y Filipič, B. (2005). Demo: Differential evolution for multiobjective optimization. En: *Proceedings of the Third International Conference on Evolutionary Multi-Criterion Optimization*. Springer-Verlag, Berlin, Heidelberg, EMO'05, pp. 520–533.

Rooy, N. (2017a). Differential evolution optimization with python. `https://github.com/nathanrooy/differential-evolution-optimization-with-python`.

Rooy, N. (2017b). Particle swarm optimization with python. `https://github.com/nathanrooy/Particle-Swarm-Optimization-with-Python`.

Saad, M. S. M., Jamaluddin, H., y Darus, I. Z. M. (2012). Implementation of PID controller tuning using differential evolution and genetic algorithms. *International Journal of Innovative Computing, Information and Control*, **8**(11): 7761–7779.

Sahib, M. A. y Ahmed, B. S. (2016). A new multiobjective performance criterion used in pid tuning optimization algorithms. *Journal of Advanced Research*, **7**(1): 125–134.

Shinners, S. M. (1998). *Modern control system theory and design*. John Wiley & Sons.

Shinskey, F. (1996). Process control systems: application, design, and tuning. mcgraw-hill co. *Inc., New York, NY*.

Siegel, S. (1956). *Nonparametric statistics for the behavioral sciences.*. McGraw-Hill. New York, NY, US, p. 312.

Skogestad, S. (2003). Simple analytic rules for model reduction and pid controller tuning. *Journal of Process Control*, **13**(4): 291–309.

Srivastava, S. y Pandit, V. (2016a). A PI/PID controller for time delay systems with desired closed loop time response and guaranteed gain and phase margins. *Journal of Process Control*, **37**: 70–77.

Srivastava, S. y Pandit, V. (2016b). Studies on pi/pid controllers in the proportional integral plane via different performance indices. En: *Control, Measurement and*

Instrumentation (CMI), 2016 IEEE First International Conference on. IEEE, pp. 151–155.

Srivastava, S., Misra, A., Thakur, S. K., y Pandit, V. S. (2016). An optimal PID controller via LQR for standard second order plus time delay systems. *ISA Transactions*, **60**: 244–253.

Steuer, R. E. (1986). Multiple criteria optimization. *Theory, Computation and Applications*.

Storn, R. y Price, K. (1997). Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, **11**(4): 341–359.

Von Lücken, C., Barán, B., y Brizuela, C. (2014). A survey on multi-objective evolutionary algorithms for many-objective problems. *Computational Optimization and Applications*, **58**(3): 707–756.

Wang, Q., Lu, C., y Pan, W. (2016). IMC PID controller tuning for stable and unstable processes with time delay. *Chemical Engineering Research and Design*, **105**(2009): 120–129.

Ziegler, J. G. y Nichols, N. B. (1942). Optimum settings for automatic controllers. *InTech*, **42**(6): 94–100.

Anexo A

A.1. Programas en Python

En este anexo se presentan los códigos en lenguaje Python de los algoritmos utilizado para realizar las simulaciones de la sintonización del controlador PID.

A.1.1. Metaheurísticas de optimización

En este apartado se muestran los códigos para la optimización mono-objetivo y multi-objetivo. Se utilizaron evolucion diferencial para los dos enfoques y optimización por enjambre de partículas para el primer enfoque.

A.1.1.1. Evolución Diferencial

```
# -*- coding: utf-8 -*-
"""
@author: rtrujillo93
"""
import random
import subprocess
import numpy as np

from time import time

#%%***** Cost Function ***** %%#
""" *****
"p", str(x[0]) = Kp
"i", str(x[1]) = Ti
"d", str(x[2]) = Td
"o", str(#) = 1 > ITAE
            = 2 > IAE
            = 3 > IE
            = 4 > ISE
            = 5 > ITSE
            = 6 > ISTAE
            = 7 > ISTSE
            = 8 > U (control signal)
From 1 to 7, it also returns overshoot, settling, L1(u), L2(u) and Linf(u)
time in the variable output
***** """
def cost_function(x):
    process = subprocess.Popen(['./progdelay',"p",str(x[0]),"i",str(x[1]),
                               "d",str(x[2]),"o",str('#')],
                               stdout=subprocess.PIPE)
    (output, err) = process.communicate()
    output = output.split()
    Linf = float(np.asarray(output[5]))         # Linf
    L2 = float(np.asarray(output[4]))           # L2
    L1 = float(np.asarray(output[3]))           # L1
    ts = float(np.asarray(output[2]))           # Settling time
    Mp = float(np.asarray(output[1]))           # Overshoot
    value = float(np.asarray(output[0]))        # Cost (Fitness)
    return (value, Mp, ts, L1, L2, Linf)
```

```
40
41 #%%***** Main ***** %%#
42 def main(S, NP, F, CR, G, cost_fn):
43
44     population_value = []           # Set population
45     overshoot_value = []            # Set overshoot
46     settlingtime_value = []         # Set settling time
47     L1_value = []                   # Set L1
48     L2_value = []                   # Set L2
49     Linf_value = []                 # Set Linf
50
51     initial_time = time ()           # Initial execution time
52
53     """ ***** Initialize population ***** """
54     population = []                 # Set population vector
55
56     for i in range(0,NP):
57         individual = []             # Set individual vector
58         for j in range(len(S)):
59             individual.append(random.uniform(S[j][0],S[j][1]))
60         population.append(individual)
61
62     """ ***** Evaluate initial population ***** """
63     for i in range(0, NP):
64         (value, Mp, ts, L1, L2, Linf) = cost_fn(population[i])
65         population_value.append(value)
66         overshoot_value.append(Mp)
67         settlingtime_value.append(ts)
68         L1_value.append(L1)
69         L2_value.append(L2)
70         Linf_value.append(Linf)
71
72     """ ****** Solution ***** """
73     for i in range(1, G + 1):
74         Mp = min(overshoot_value)           # Overshoot restriction
75         ts = min(settlingtime_value)        # Settling time restriction
76
77         for j in range(0, NP):
78
79             """ ***** Mutation ***** """
80             """ There are two variants, one where the objective vector
81             is random and the other, where each of the individuals of
82             the population is used. """
83             random_vectors = range(0, NP)
84             """ if choose random target vector comment this line """
85             random_vectors.remove(j)
86             """ if random target vector choose 4, else 3 """
87             random_index = random.sample(random_vectors, 3)
88
89             x1 = population[random_index[0]]
90             x2 = population[random_index[1]]
91             x3 = population[random_index[2]]
92 #            target_vector = population[random_index[3]]    # Random target vector
93             target_vector = population[j]       # Target vector
94
95             difference = [x2_i - x3_i for
96                 x2_i, x3_i in zip(x2, x3)]      # Difference (x2 - x3)
97
98             mutant_vector = [x1_i + F * difference_i for
99                 x1_i, difference_i in
100                zip(x1, difference)]           # x1 + F * difference
```

```
101
102             """ ***** Crossover ***** """
103             trial_vector = []
104
105             for k in range(len(target_vector)):
106                 crossover = random.random()
107                 rnbr = random.randint(0, 2)
108                 if crossover <= CR or rnbr == k:
109                     trial_vector.append(mutant_vector[k])
110                 else:
111                     if crossover > CR and rnbr != k:
112                         trial_vector.append(target_vector[k])
113
114             """ Selection (Greedy criterion)"""
115             (trial_value, trial_overshoot,
116                  trial_settlingtime, trial_L1,
117                  trial_L2, trial_Linf) = cost_fn(trial_vector)
118
119             (target_value, target_overshoot,
120                  target_settlingtime, target_L1,
121                  target_L2, target_Linf) = cost_fn(target_vector)
122
123             if (trial_value < target_value and trial_overshoot <= Mp
124                 and trial_settlingtime <= ts):
125                 population[j] = trial_vector  # [random_index[3]]
126                 population_value[j] = trial_value
127                 overshoot_value[j] = trial_overshoot
128                 settlingtime_value[j] = trial_settlingtime
129                 L1_value[j] = trial_L1
130                 L2_value[j] = trial_L2
131                 Linf_value[j] = trial_Linf
132             else:
133                 if target_overshoot <= Mp:
134                     population[j] = target_vector
135                     population_value[j] = target_value
136                     overshoot_value[j] = target_overshoot
137                     settlingtime_value[j] = target_settlingtime
138                     L1_value[j] = target_L1
139                     L2_value[j] = target_L2
140                     Linf_value[j] = target_Linf
141                 else:
142                     population[j] = trial_vector  # [random_index[3]]
143                     population_value[j] = trial_value
144                     overshoot_value[j] = trial_overshoot
145                     settlingtime_value[j] = trial_settlingtime
146                     L1_value[j] = trial_L1
147                     L2_value[j] = trial_L2
148                     Linf_value[j] = trial_Linf
149         final_time = time()
150
151     execution_time = final_time - initial_time
152
153     solution = population[population_value.index(min(population_value))]
154     value_cost_fn = min(population_value)
155     overshoot_solution = overshoot_value[population_value.
156         index(min(population_value))]
157     settlingtime_solution = settlingtime_value[population_value.
158         index(min(population_value))]
159     L1_solution = L1_value[population_value.index(min(population_value))]
160     L2_solution = L2_value[population_value.index(min(population_value))]
161     Linf_solution = Linf_value[population_value.index(min(population_value))]
```

```
162
163     print solution[0],
164     print solution[1],
165     print solution[2],
166     print value_cost_fn,
167     print overshoot_solution,
168     print settlingtime_solution,
169     print L1_solution,
170     print L2_solution,
171     print Linf_solution,
172     print execution_time
173
174     return solution
175 #%%***** Variables ***** %%#
176 S = [(0.0, 10.0), (0.0, 1.0), (0.0, 0.2)]   # Search space [Kp, Ti, Td]
177 NP = 30                      # Population size = (10 o 15) * len(S)
178 G = 50                       # Generations
179 F = 0.5                      # Mutation constant  [0,2]
180 CR = 0.85                    # Crossover constant [0,1]
181 cost_fn = cost_function      # Cost function
182
183 #%%****** Run ***** %%#
184 main(S, NP, F, CR, G, cost_fn)
```

A.1.1.2. Optimización por enjambre de partículas

```
1  # -*- coding: utf-8 -*-
2  """
3  @author: rtrujillo93
4  """
5  import random
6  import subprocess
7  import numpy as np
8
9  from time import time
10
11 #%%***** Cost Function ***** %%#
12 """ *****
13 "p", str(x[0]) = Kp
14 "i", str(x[1]) = Ti
15 "d", str(x[2]) = Td
16 "o", str(#) = 1 > ITAE
17             = 2 > IAE
18             = 3 > IE
19             = 4 > ISE
20             = 5 > ITSE
21             = 6 > ISTAE
22             = 7 > ISTSE
23             = 8 > NormaL1
24             = 9 > NormaL2
25             = 10 > U (control signal)
26 From 1 to 7, it also returns overshoot ans settling time in the variable
27 output
28 ***** """
29 def cost_function(x):
30     process = subprocess.Popen(['./progdelay',"p",str(x[0]),"i",str(x[1]),
31                                 "d",str(x[2]),"o",str('#')],
32                                 stdout=subprocess.PIPE)
33     (output, err) = process.communicate()
34     output = output.split()
35     Linf = float(np.asarray(output[5]))        # Linf
```

```
    L2 = float(np.asarray(output[4]))          # L2
    L1 = float(np.asarray(output[3]))          # L1
    ts = float(np.asarray(output[2]))          # Settling time
    Mp = float(np.asarray(output[1]))          # Overshoot
    value = float(np.asarray(output[0]))       # Cost (Fitness)
    return (value, Mp, ts, L1, L2, Linf)

#%%***** Particle Class ***** #
class Particle:
    def __init__(self):
        self.position = []          # Position
        self.velocity = []          # Velocity
        self.p_best = []            # Better individual position
        self.g_best = -1            # Best individual cost
        self.value = -1             # Individual cost
        self.Mp = -1                # Overshoot
        self.ts = -1                # Settling time
        self.L1 = -1                # Norm L1 control
        self.L2 = -1                # Norm L2 control
        self.Linf = -1              # Norm Linf control

        """ ***** Initialization of particles with random position
        and speed ***** """
        for i in range(0, D):
            self.velocity.append(random.uniform(0, 0))
            self.position .append(random.uniform(S[i][0], S[i][1]))

    """ ***** Evaluation to obtain fitness ***** """
    def Evaluate(self, c_function):
        (self.value, self.Mp, self.ts, self.L1,
             self.L2, self.Linf) = c_function(self.position)

        if self.value < self.g_best or self.g_best == -1:
            self.p_best = self.position
            self.g_best = self.value

    """ ***** Velocity update ***** """
    def Velocity_update(self,gp_best):
        w = 0.9       # Weight of inertia
        c1 = 2        # cognitive constant
        c2 = 2        # Social constant

        for i in range(0, D):
            r1 = random.random()
            r2 = random.random()

            self.velocity[i] = w*self.velocity[i] + c1*r1*(self.p_best[i]-
                self.position [i]) + c2*r2*(gp_best[i]-self.position [i])

    """ ***** Position update ***** """
    def Position_update(self, S):
        for i in range(0, D):
            self.position [i] = self.position [i] + self.velocity[i]

#%%***** Main ***** %%#
class PSO():
    def __init__(self, c_function, S, NP, M):
        global D
        global Mps

        initial_time = time()
```

```
97
98          D = len(S)          # Space dimension
99          gc_best = -1        # Better global cost
100         gp_best = []        # Better global position
101         Mp = -1             # Overshoot
102         ts = -1             # Settling time
103         L1 = -1                # Norm L1 control
104         L2 = -1                # Norm L2 control
105         Linf = -1              # Norm Linf control
106
107         """ ***** Initialization of the swarm ***** """
108         global swarm
109         swarm = []
110         minmp = []
111         mints = []
112         for i in range(0, NP):
113             swarm.append(Particle())
114
115         """ ***** Solution ***** """
116         i = 1
117         while i < M+1:
118
119             """ ***** Evaluate fitness ***** """
120             for j in range(0, NP):
121                 swarm[j].Evaluate(c_function)
122
123                 minmp.append(swarm[j].Mp)
124                 mints.append(swarm[j].ts)
125                 Mps = min(minmp)        # Restricted overshoot
126                 tss = min(mints)        # Restricted settling time
127                 """ ***** Determine gbest ***** """
128                 if swarm[j].value < gc_best or gc_best == -1:
129                     if swarm[j].Mp <= Mps and swarm[j].ts <= tss:
130                         gp_best = list(swarm[j].position)
131                         gc_best = float(swarm[j].value)
132                         Mp = float(swarm[j].Mp)
133                         ts = float(swarm[j].ts)
134                         L1 = float(swarm[j].L1)
135                         L2 = float(swarm[j].L2)
136                         Linf = float(swarm[j].Linf)
137
138             """ ***** Update speeds and positions ***** """
139             for j in range(0, NP):
140                 swarm[j].Velocity_update(gp_best)
141                 swarm[j].Position_update(S)
142             i = i + 1
143
144         final_time = time()
145         execution_time = final_time - initial_time
146
147 #        # ***** Results ***** #
148         print gp_best[0],               # KP
149         print gp_best[1],               # Ti
150         print gp_best[2],               # Td
151         print gc_best,                  # Value
152         print Mp,                       # Overshoot
153         print '%.3f' %ts,              # Settling time
154         print '%.2f' %L1,              # L1
155         print '%.2f' %L2,              # L2
156         print '%.2f' %Linf,            # Linf
157         print '%.2f' %execution_time   # Execution time
```

```
158
159 #%%***** Variables ***** %%#
160 S = [(0.0, 10.0), (0.0, 1.0), (0.0, 0.2)]       # Search space [Kp, Ti, Td]
161 NP = 30          # Population size
162 M = 85           # Iterations
163
164 #%%***** Run ***** %%#
165 PSO(cost_function, S, NP, M)        # Inicio del programa
```

A.1.1.3. Evolución diferencial para la optimizacin multi-objetivo (2 criterios)

Para implementar la optimización multi-objetivo se utiliza gran parte del código presentado en el anexo A.1.1.1, así como del siguiente paquete.

```
1 import pygmo as pg
```

Este paquete es de utilidad para obtener los frentes del conjunto de posibles soluciones y de ellos obtener el frente no dominado. A continuación el código se muestra a partir de la selección, ya que es donde ocurre lo más importante de este enfoque.

```
1             """ ***** Selection (Greedy criterion) ***** """
2             (trial_overshoot, trial_settlingtime, trial_L1,
3                  trial_L2, trial_Linf) = cost_fn(trial_vector)
4             (target_overshoot, target_settlingtime, target_L1,
5                  target_L2, target_Linf) = cost_fn(target_vector)
6
7             if (trial_overshoot < target_overshoot
8                     and trial_settlingtime < target_settlingtime):
9                 population[j] = trial_vector
10                overshoot_value[j] = trial_overshoot
11                settlingtime_value[j] = trial_settlingtime
12                L1_value[j] = trial_L1
13                L2_value[j] = trial_L2
14                Linf_value[j] = trial_Linf
15            else:
16                if (trial_overshoot < target_overshoot
17                        and trial_settlingtime > target_settlingtime
18                        or trial_overshoot > target_overshoot
19                        and trial_settlingtime < target_settlingtime):
20                    population.append(trial_vector)
21                    overshoot_value.append(trial_overshoot)
22                    settlingtime_value.append(trial_settlingtime)
23                    L1_value.append(trial_L1)
24                    L2_value.append(trial_L2)
25                    Linf_value.append(trial_Linf)
26
27        """ ***** Truncate ***** """
28        if len(population) > NP:
29            data = []
30            full_data = []
31            for l in range(0,len(population)):
32                data.append((settlingtime_value[l], overshoot_value[l]))
33                full_data.append((settlingtime_value[l], overshoot_value[l],
34                                  L1_value[l], L2_value[l], Linf_value[l]))
35            """ ****** non-dominated sorting ***** """
36            ndf, dl, dc, ndr = pg.fast_non_dominated_sorting(points = data)
37            bestN = pg.select_best_N_mo(data, NP)
38            population_aux = population
```

```
            overshoot_aux = overshoot_value
            settlingtime_aux = settlingtime_value
            L1_aux = L1_value
            L2_aux = L2_value
            Linf_aux = Linf_value
            data_aux = data
            full_aux = full_data
            m = 0
            while m < len(population):
                if m < NP:
                    population[m] = population_aux[bestN[m]]
                    overshoot_value[m] = overshoot_aux[bestN[m]]
                    settlingtime_value[m] = settlingtime_aux[bestN[m]]
                    L1_value[m] = L1_aux[bestN[m]]
                    L2_value[m] = L2_aux[bestN[m]]
                    Linf_value[m] = Linf_aux[bestN[m]]
                    full_data[m] = full_aux[bestN[m]]
                    data[m] = data_aux[bestN[m]]
                    m = m+1
                else:
                    del overshoot_value[m:len(population)+1]
                    del settlingtime_value[m:len(population)+1]
                    del L1_value[m:len(population)+1]
                    del L2_value[m:len(population)+1]
                    del Linf_value[m:len(population)+1]
                    del full_data[m:len(population)+1]
                    del data[m:len(population)+1]
                    del population[m:len(population)+1]
                    m = len(population)
        population_aux = population
        overshoot_aux = overshoot_value
        settlingtime_aux = settlingtime_value
        L1_aux = L1_value
        L2_aux = L2_value
        Linf_aux = Linf_value
        data_aux = data
        full_aux = full_data
        shuffle_enumerate = range(0, NP)
        random_index = random.sample(shuffle_enumerate, NP)
        for n in range(0,len(random_index)):
            population[n] = population_aux[random_index[n]]
            overshoot_value[n] = overshoot_aux[random_index[n]]
            settlingtime_value[n] = settlingtime_aux[random_index[n]]
            L1_value[n] = L1_aux[random_index[n]]
            L2_value[n] = L2_aux[random_index[n]]
            Linf_value[n] = Linf_aux[random_index[n]]
            full_data[n] = full_aux[random_index[n]]
            data[n] = data_aux[random_index[n]]
            ndf, dl, dc, ndr = pg.fast_non_dominated_sorting(points = data)
            bestN = pg.select_best_N_mo(data, NP)

        global final_time
        final_time = time()

    execution_time = final_time - initial_time

    print 'Execution time: ',execution_time

    for i in range(0,len(population)):
        print population[i][0],
        print population[i][1],
```

```
100         print population[i][2],
101         print '%.2f' %overshoot_value[i],
102         print '%.3f' %settlingtime_value[i],
103         print '%.2f' %L1_value[i],
104         print '%.2f' %L2_value[i],
105         print '%.2f' %Linf_value[i]
106 
107     return population
108 
109 #%%***** Variables ***** %%#
110 S = [(0.0, 10.0), (0.0, 1.0), (0.0, 0.2)]   # Search space [Kp, Ti, Td]
111 NP = 30                     # Population size = (10 o 15) * len(S)
112 G = 50                      # Generations
113 F = 0.5                     # Mutation constant  [0,2]
114 CR = 0.85                   # Crossover constant [0,1]
115 cost_fn = cost_function     # Cost function
116 
117 #%%****** Run ***** %%#
118 main(S, NP, F, CR, G, cost_fn)
```

A.1.1.4. Evolución diferencial para la optimizacin multi-objetivo (3 criterios)

Para tres criterios prácticamente es el mismo código al presentado en el anexo A.1.1.3 salvo al momento de realizar la selección, a continuación se muestra parte de la modificación que se debe de hacer para optimizar con dicha cantidad de criterios.

```
1             """ ***** Selection (Greedy criterion) ***** """
2             (trial_overshoot, trial_settlingtime, trial_L1,
3                  trial_L2, trial_Linf) = cost_fn(trial_vector)
4             (target_overshoot, target_settlingtime, target_L1,
5                  target_L2, target_Linf) = cost_fn(target_vector)
6 
7             if (trial_overshoot < target_overshoot
8                     and trial_settlingtime < target_settlingtime
9                     and trial_Linf < target_Linf):
10                population[j] = trial_vector
11                overshoot_value[j] = trial_overshoot
12                settlingtime_value[j] = trial_settlingtime
13                L1_value[j] = trial_L1
14                L2_value[j] = trial_L2
15                Linf_value[j] = trial_Linf
16            else:
17                if (target_overshoot < trial_overshoot
18                        and target_settlingtime < trial_settlingtime
19                        and target_Linf < trial_Linf):
20                    pass
21                else:
22                    population.append(trial_vector)
23                    overshoot_value.append(trial_overshoot)
24                    settlingtime_value.append(trial_settlingtime)
25                    L1_value.append(trial_L1)
26                    L2_value.append(trial_L2)
27                    Linf_value.append(trial_Linf)
```

A.1.2. Comunicación entre Python y C++

Todo los algoritmos presentados hasta el momento tienen en común una parte del código, que es el que se encarga de enviarle los parámetros del controlador PID

al subprograma códificado en lenguaje C++, que es el que realiza la simulación del sistema con retardo. Esto se lleva a cabo mediante el uso del siguiente paquete.

```
import subprocess
```

Mediante el siguiente código se lleva a cabo la comunicación entre los dos lenguajes.

```
#%%***** Cost Function ***** %%#
""" *****
"p", str(x[0]) = Kp
"i", str(x[1]) = Ti
"d", str(x[2]) = Td
"o", str(#) = 1 > ITAE
            = 2 > IAE
            = 3 > IE
            = 4 > ISE
            = 5 > ITSE
            = 6 > ISTAE
            = 7 > ISTSE
            = 8 > U (control signal)
From 1 to 7, it also returns overshoot, settling, L1(u), L2(u) and Linf(u)
time in the variable output
***** """
def cost_function(x):
    process = subprocess.Popen(['./progdelay',"p",str(x[0]),"i",str(x[1]),
                               "d",str(x[2]),"o",str('#')],
                               stdout=subprocess.PIPE)
    (output, err) = process.communicate()
    output = output.split()
    Linf = float(np.asarray(output[5]))          # Linf
    L2 = float(np.asarray(output[4]))            # L2
    L1 = float(np.asarray(output[3]))            # L1
    ts = float(np.asarray(output[2]))            # Settling time
    Mp = float(np.asarray(output[1]))            # Overshoot
    return (
    Mp, ts, L1, L2, Linf)
```

A.1.3. Obtención de código en lenguaje C++

Una parte fundamental de la simulación, es donde se lleva a cabo la solución del modelo del sistema de control, el paquete `pydelay`, además de implementarse en lenguaje Python, ofrece la posibilidad de obtener el código en lenguaje C++. Esto conlleva ventajas sobre todo en tiempo de ejecución y uso de recursos de la computadora. El siguiente código muestra un ejemplo para simular del sistema con retardo que se manejó en esta tesis.

```
# -*- coding: utf-8 -*-
```

```
2
3 from pydelay import dde23
4
5 tfinal = 2.5
6
7 ## ********** System state-space model *********** #
8 # Set equations
9 eqns = {
10         'x1' : 'x2',
11         'x2' : '(-0.68*x2 - 1.45*x1 + 0.5*(Kp*((1-x1(t-L)) - Td*x2(t-L) + (1/Ti)*z)))
       /0.08',
12         'z' : '(1-x1(t-L))',
13         'u' : 'Kp*((1-x1(t-L)) - Td*x2(t-L) + (1/Ti)*z)',}
14
15 # Set parametric values
16 params={
17         'L': L,
18         'Kp': kp,
19         'Ti': ti,
20         'Td': td,
21   }
22
23 # Initialize solver
24 dde = dde23(eqns=eqns, params=params)
25
26 # Set simulation parameters
27 # (Solve for t=0 to t=2.5 with step 0.01)
28 dde.set_sim_params(tfinal, AbsTol = 1.0e-6, RelTol = 1.0e-5,dtmin=0.01, dtmax=0.01, dt0
       =0.01)
29
30 # Set initial conditions
31 dde.hist_from_funcs({'x1': lambda t: 0.0,
32                      'x2': lambda t: 0.0,
33                      'z': lambda t: 0.0,
34                      'u': lambda t: 0.0,})
35
36 # Get code in C++ of the system with delay
37 #print(dde.output_ccode())
38
39 # Run Solver
40 dde.run()
41
42 # Solution with a sample size dt=0.01 between 0 and 2.5.
43 sol = dde.sample(0, tfinal, 0.01)
44
45
46 # Solution
47 y = sol['x1']
48 t = sol['t']
```

Para obtener el programa en lenguaje C++ solo es necesario descomentar una línea de código, la siguiente.

```
1 # Get code in C++ of the system with delay
2 print(dde.output_ccode())
```

En el subsección B.1 se muestra el manejo que debe realizarse con el código en C++, para llevar a cabo correctamente las simulaciones.

Anexo B

En este anexo se muestran los códigos en lenguaje C++ utilizados para realizar la simulación de los sistemas de control con retardo.

B.1. Programa en C++ y modificaciones al código

El código persé no funciona correctamente, así que debe de hacerse una serie de modificaciones para adecuarlo para las simulaciones en comunicación con el programa en lenguaje Python. Para ello, a continuación se enlistan todos los cambios pertinentes.

1. Es necesario descomentar la siguiente línea.

```
//#define MANUAL
```

```
#define MANUAL
```

2. La función `main` por defecto tiene esto.

```
int main()
{
```

   Hay que cambiarlo por lo siguiente.

```
int main(int argc, char* argv[])
{
```

   Como se puede observar la función `main()` ahora tien los argumentos `int argc, char* argv[]`, estos sirven para poder recibir los parámetros que envía el programa de optimización en Python.

3. Los parámetros para ajustar el solucionador presenta por defecto estos valores.

```
double dtmin = 0.0001;
double dtmax = 0.1;
double tfinal = 100.0;
double RelTol = 1.0E-3;
double AbsTol = 1.0E-6;
int chunk = 10000;
int nstart = 101;
double dt0 = 0.01;
double maxdelay = ...; // Set the maximum delay here !!!
long unsigned int MaxIter = 10000000;
int NumOfDiscont = 4;
double discont[4] = {maxdelay, 2*maxdelay, 3*maxdelay, tfinal};
```

   Lo mas adecuado es utilizar lo siguiente.

```
double dtmin = 0.01;
double dtmax = 0.01;
double tfinal = 25.0;
double RelTol = 1.0E-6;
double AbsTol = 1.0E-6;
int chunk = 10000;
int nstart = 21;
double dt0 = 0.01;
double maxdelay = 0.2; // Set the maximum delay here !!!
long unsigned int MaxIter = 10000000;
int NumOfDiscont = 4;
double discont[4] = {maxdelay, 2*maxdelay, 3*maxdelay, tfinal};
double PARL = 0.2;
```

Los parámetros `dtmin` y `dtmax` indican el paso mínimo y máximo, en este caso se utilizó 0.01 en ambos, `tfinal` representa el tiempo de simulación, el cual fue de 25 segundos, `Reltol` y `Abstol` son la tolerancia relativa y absoluta, se utilizó un valor de 1.0E-6 para ambos, la variable chunk se deja con su valor por defecto, la variable `nstart` se define como el retardo entre el paso más uno, es decir, si se utiliza un retardo de 0.2 segundos, $nstart = \frac{0.2}{0.01} + 1$, el parámetro `dt0` lo igualamos a `dtmin`, es decir, a 0.01 segundos, `maxdelay` se iguala al valor del retardo utilizado, en este modelo es de 0.2 segundos. Las variables `MaxIter`, `NumOfDiscont` y `discont[4]`, se dejan por defecto. Por último `ParL` es igual al valor del retardo, 0.2 segundos.

4. Los valores de los parámetros involucrados en el sistema de ecuaciones del sistema, por defecto se declaran de la siguiente forma dentro de la función `main()`.

```
double PARTd = 0.9725977835;
double PARKp = 5.26553022244;
double PARTi = 0.195527660669;
```

En este caso como se necesitaba tener que mandar seguidamente distintos conjuntos de parámetros se utilizó un subprograma llamado `programas.cpp` que se agrega inmediatamente despues de la función `main()`, y se muestra a continuación.

```
double PARKp;
double PARTd;
double PARTi;
int opc;
```

```
5
6   if (argc > 1)
7   {
8     for(int i = 1; i<argc; i+=2)
9     {
10        switch(char(argv[i][0]))
11        {
12      case 'p':
13      {
14          PARKp = atof(argv[i+1]);
15      }
16          break;
17      case 'i':
18      {
19          PARTi = atof(argv[i+1]);
20      }
21          break;
22      case 'd':
23      {
24          PARTd = atof(argv[i+1]);
25      }
26          break;
27      case 'o':
28      {
29          opc = atof(argv[i+1]);
30      }
31          break;
32        } // end switch
33    } // end for
34  } // end if
```

```
1 int main(int argc, char* argv[])
2 {
3 #include "parametros.cpp"
```

5. La línea que se muestra a continuación debe comentarse para que al compilar el programa no se generen errores.

```
1 srand((unsigned)RSEED);
```

```
1 //srand((unsigned)RSEED);
```

6. Las condiciones iniciales por defecto vienen inicializadas con un valor de 0.2.

```
1 #ifdef MANUAL
2 for(i = 0; i < nstart+1; i++)
3     pt_t_ar[i]  = -maxdelay*(nstart-i)/nstart;
4
5 /* set the history here when running generated code directly */
6
7 for(i = 0; i < nstart+1; i++) {
8     pt_x2_ar[i]  = 0.2; // history value for the variable
9     pt_x2_Var[i] = 0.0; // history of the derivatives (0.0 if constant history)
10 }
11
```

```
for(i = 0; i < nstart+1; i++) {
    pt_x1_ar[i]  = 0.2; // history value for the variable
    pt_x1_Var[i] = 0.0; // history of the derivatives (0.0 if constant history)
}

for(i = 0; i < nstart+1; i++) {
    pt_u_ar[i]  = 0.2; // history value for the variable
    pt_u_Var[i] = 0.0; // history of the derivatives (0.0 if constant history)
}

for(i = 0; i < nstart+1; i++) {
    pt_z_ar[i]  = 0.2; // history value for the variable
    pt_z_Var[i] = 0.0; // history of the derivatives (0.0 if constant history)
}

#endif
```

En las simulaciones se necesita un valor de 0.0.

```
#ifdef MANUAL
for(i = 0; i < nstart+1; i++)
    pt_t_ar[i]  = -maxdelay*(nstart-i)/nstart;

/* set the history here when running generated code directly */

for(i = 0; i < nstart+1; i++) {
    pt_x2_ar[i]  = 0.0; // history value for the variable
    pt_x2_Var[i] = 0.0; // history of the derivatives (0.0 if constant history)
}

for(i = 0; i < nstart+1; i++) {
    pt_x1_ar[i]  = 0.0; // history value for the variable
    pt_x1_Var[i] = 0.0; // history of the derivatives (0.0 if constant history)
}

for(i = 0; i < nstart+1; i++) {
    pt_u_ar[i]  = 0.0; // history value for the variable
    pt_u_Var[i] = 0.0; // history of the derivatives (0.0 if constant history)
}

for(i = 0; i < nstart+1; i++) {
    pt_z_ar[i]  = 0.0; // history value for the variable
    pt_z_Var[i] = 0.0; // history of the derivatives (0.0 if constant history)
}

#endif
```

7. Por último se muestra la siguiente parte del código donde se realiza la impresión en pantalla de la serie de tiempo, la respuesta del sistema y la señal de control del sistema de control con retardo simulado.

```
#ifdef MANUAL
for(i = 0; i < SIM_n; i++)
    std:: cout << pt_t_ar[i] << " " << pt_x2_ar[i] << " " << pt_x1_ar[i] << " " <<
     pt_u_Var[i] << " " << pt_z_ar[i] << std::endl;
```

```
    return 0;
}
#endif
```

Esto se cambio por un archivo llamado `print.cpp`, el cual funciona con base a la opción que se haya asignado en los parámetros que se reciben en el punto 4.

```
#ifdef MANUAL
#include "print.cpp"
    return 0;
}
#endif
```

El archivo `print.cpp` contiene lo siguiente.

```
switch(opc){
  case 1: std:: cout << ITAE(dt,SIM_n,pt_t_ar,pt_x1_ar) << std::endl;
      std:: cout << OS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << TS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << NormaL1(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaL2(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaLinf(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
    break;
  case 2: std:: cout << IAE(dt,SIM_n,pt_t_ar,pt_x1_ar) << std::endl;
        std:: cout << OS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << TS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << NormaL1(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaL2(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaLinf(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;

    break;
  case 3: std:: cout << IE(dt,SIM_n,pt_t_ar,pt_x1_ar) << std::endl;
      std:: cout << OS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << TS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << NormaL1(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaL2(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaLinf(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
    break;
  case 4: std:: cout << ISE(dt,SIM_n,pt_t_ar,pt_x1_ar) << std::endl;
        std:: cout << OS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << TS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << NormaL1(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaL2(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaLinf(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
    break;
  case 5: std:: cout << ITSE(dt,SIM_n,pt_t_ar,pt_x1_ar) << std::endl;
      std:: cout << OS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << TS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << NormaL1(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaL2(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaLinf(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
    break;
  case 6: std:: cout << ISTAE(dt,SIM_n,pt_t_ar,pt_x1_ar) << std::endl;
        std:: cout << OS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << TS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << NormaL1(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaL2(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
```

```
      std:: cout << NormaLinf(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
    break;
  case 7: std:: cout << ISTSE(dt,SIM_n,pt_t_ar,pt_x1_ar) << std::endl;
      std:: cout << OS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << TS(SIM_n, pt_t_ar, pt_x1_ar) << std::endl;
      std:: cout << NormaL1(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaL2(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
      std:: cout << NormaLinf(dt,SIM_n,pt_t_ar,pt_u_Var) << std::endl;
    break;
  case 8: U(SIM_n, pt_t_ar, pt_u_Var);
    break;
}
```

8. Cada cálculo de los criterios o funciones objetivo además de las normas de control fue codificado en lenguaje C++, el nombre del archivo para esto se llama `criterios.cpp` y contiene lo siguiente.

```
#include "criterios.h"
#include<cmath>

#include <iostream>

double ITAE(double dt, long unsigned int n, double * ts, double * xs){
    long double sumITAE = 0.0;
    long unsigned int i;
    for(i = 1; i < n; i++)
        if (ts[i] > 0.0)
    sumITAE += (ts[i]*std::abs(1.0-xs[i]) + ts[i-1]*std::abs(1.0-xs[i-1]));
    return(dt*sumITAE/2.0);
}

double IAE(double dt, long unsigned int n, double * ts, double * xs){
    double sumIAE=0.0;
    long unsigned int i;
    for(i = 1; i < n; i++)
        if (ts[i] > 0.0)
          sumIAE += (std::abs(1.0-xs[i]) + std::abs(1.0-xs[i-1]));
    return(dt*sumIAE/2.0);
}

double IE(double dt, long unsigned int n, double * ts, double * xs){
  double sumIE = 0.0;
  long unsigned int i;
  for(i = 1; i < n; i++)
    if (ts[i] > 0.0)
      sumIE += ((1.0-xs[i]) + (1.0-xs[i-1]));
  return(std::abs(dt*sumIE/2.0));
}

double ISE(double dt, long unsigned int n, double * ts, double * xs){
  double sumISE = 0.0;
  long unsigned int i;
  for(i = 1; i < n; i++)
    if (ts[i] > 0.0)
      sumISE += pow((1.0-xs[i]),2.0) + pow((1.0-xs[i-1]),2.0);
  return(dt*sumISE/2.0);
}
```

```
41
42 double ITSE(double dt, long unsigned int n, double * ts, double * xs){
43   double sumITSE = 0.0;
44   long unsigned int i;
45   for(i = 1; i < n; i++)
46     if (ts[i] > 0.0)
47       sumITSE += ts[i] * pow((1.0-xs[i]), 2.0) + ts[i-1] * pow((1.0-xs[i-1]), 2.0)
     ;
48   return(dt*sumITSE/2.0);
49 }
50
51 double ISTAE(double dt, long unsigned int n, double * ts, double * xs){
52   double sumISTAE = 0.0;
53   long unsigned int i;
54   for(i = 1; i < n; i++)
55     if (ts[i] > 0.0)
56       sumISTAE += pow(ts[i],2.0) * std::abs(1.0-xs[i]) + pow(ts[i-1],2.0) * std::
     abs(1.0-xs[i-1]);
57   return(dt*sumISTAE/2.0);
58 }
59
60 double ISTSE(double dt, long unsigned int n, double * ts, double * xs){
61   double sumISTSE = 0.0;
62   long unsigned int i;
63   for(i = 1; i < n; i++)
64     if (ts[i] > 0.0)
65       sumISTSE += pow(ts[i],2.0) * pow((1.0-xs[i]),2.0) + pow(ts[i-1],2.0) * pow
     ((1.0-xs[i-1]),2.0);
66   return(dt*sumISTSE/2.0);
67 }
68
69 double NormaL1(double dt, long unsigned int n, double * ts, double * us){
70     double L1=0.0;
71     long unsigned int i;
72     for(i = 1; i < n; i++)
73         if (ts[i]>0.0)
74             L1 += (std::abs(us[i])+std::abs(us[i-1]));
75     return(dt*L1/2.0);
76 }
77
78 double NormaL2(double dt, long unsigned int n, double * ts, double * us){
79     double L2=0.0;
80     long unsigned int i;
81     for(i = 1; i < n; i++)
82     if (ts[i]>0.0)
83             L2 += pow(pow((std::abs(us[i])+std::abs(us[i-1])),2.0), 0.5);
84     return(pow(dt, 2.0)*L2/4.0);
85     // \sum (dt*(|x[i]| + |x[i-1]|)/2))^2 = dt^2* (\sum (|x[i]| + |x[i-1]|)^2 )/ 4
86 }
87
88 double NormaLinf(double dt, long unsigned int n, double * ts, double * us){
89     double Linf=0.0;
90     long unsigned int i;
91     for(i = 1; i < n; i++)
92     if (ts[i]>0.0)
93           if(std::abs(us[i]) > Linf)
94         Linf = us[i];
95     return(Linf);
96 }
```

9. El cálculo de la señal de control está codificada en C++, y en vez de dar un valor se guarda toda la serie de tiempo y la señal de control en un archivo de texto. El código del archico `control.cpp` es el siguiente.

```
#include "control.h"
#include <iostream>
#include <fstream>

using namespace std;

int U(long unsigned int n, double * ts, double * us){

long unsigned int i;

ofstream myfile;
myfile.open ("ucontrol.txt");
  for(i = 1; i < n; i++)
          if (ts[i] >= 0.0){
        myfile << ts[i] << "," << us[i] << "\n";
    }
myfile.close();
return 0;
}

```

10. El cálculo del sobre impulso y tiempo de asentamiento de igual forma se codificaron en C++. Los nombres de los archivos son `overshoot.cpp` y `settlingtime.cpp` y se muestran a continuación.

```
#include "overshoot.h"
#include <iostream>

double OS(long unsigned int n, double * ts, double * xs)
{
long double overshoot = 1.00;
long unsigned int i;

  for(i = 1; i < n; i++)
    if (ts[i] > 0.0)
      if(xs[i] > overshoot)
        overshoot  = xs[i];

  return(((overshoot-1.00)*100.00));
}
```

```
#include "settlingtime.h"
#include <iostream>

double TS(long unsigned int n, double * ts, double * xs)
{
long double stinf = 0.98;
long double stsup = 1.02;
long double settlingtime = 0.00;
long unsigned int i;
```

```
10
11    for(i = 1; i < n; i++)
12      if (ts[i] > 0.0)
13        if (xs[i] <= stinf || xs[i] >= stsup)
14            settlingtime  = ts[i];
15
16    return(settlingtime);
17  }
```

11. Cada subprograma `.cpp` lleva su correspondiente cabecera `.h`. Para compilar cada subprograma es necesario ejecutar el siguiente comando en terminal.

```
1 g++ -c subprograma.cpp
```

Como resultado se tiene un archivo con el nombre con la extensión `subprograma.o`.

12. Por último para crear el ejecutable que se utiliza en el código en Python, deben compilarse todos los subprogramas con el programa principal del sistema con retardo con el siguiente comando.

```
1 g++ -lm -o nombre_del_ejecutable nombre_del_programa_principal.cpp subprograma1.o
    subprograma2.o subprogramaN.o
```