Curso 1995/96
CIENCIAS Y TECNOLOGÍAS

ALBERTO FRANCISCO HAMILTON CASTRO

Estrategias de control óptimo basadas en programación dinámica y redes neuronales para sistemas MIMO continuos no lineales

Directores
LORENZO MORENO RUIZ
LEOPOLDO ACOSTA SÁNCHEZ

SERVICIO DE PUBLICACIONES
UNIVERSIDAD DE LA LAGUNA

SOPORTES AUDIOVISUALES E INFORMÁTICOS
Serie Tesis Doctorales

A mi esposa
Mary Carmen
por su amor
y constante apoyo.

A mis padres
con cariño.

Quiero expresar mi agradecimiento, en primer lugar, al Dr. D. Lorenzo Moreno Ruiz por su acertada dirección y colaboración, y por haberme infundido su inagotable espíritu universitario. En segundo lugar, aunque no en menor medida, al Dr. D. Leopoldo Acosta Sánchez, por la preocupación y ánimo puesto en este trabajo, así como por sus buenas ideas.

Quiero agradecer a D. José Demetrio Piñeiro Vera su compañerismo y ayuda generosa, constante y desinteresada.

A los restantes miembros del grupo de Computadoras y Control: D. José Luis Sánchez de la Rosa, D. Juan Alvino Méndez Pérez, D. Juan Julián Merino Rubio, D. Nicolás Marichal Plasencia, Da. Rosa María Aguliar Chinea, D. José Sigut Saavedra, D. José Ignacio Estévez Damas y Dr. D. Gumersindo de Vera Casanova, por su aliento y apoyo. A D. Juan Ramón Rodríguez Santana y D. Roberto Betancor Bonilla por su diligencia y disponibilidad.

Quiero agradecer también a todos y cada uno de los profesores, tanto de la Universidad de La Laguna como de la Universidad Complutense de Madrid, que me brindaron la posibilidad de formarme como Físico e Informático.

Quiero agradecer a mi esposa Mary Carmen su ayuda, entre otras muchísimas cosas, en la escritura, confección y corrección de esta memoria.

Un muy especial agradecimiento a D. Hugo C. Hamilton Hernández, mi padre, porque él, con su ejemplo y sus fértiles enseñanzas, puso una sólida base a lo que he llegado y podré llegar a ser.

Introducción

Este trabajo se enmarca dentro de la línea de investigación, en temas de control óptimo, que se viene desarrollando en el grupo de Computadoras y Control del departamento de Física Fundamental y Experimental de la Universidad de La Laguna. Inicialmente surge para cubrir una de las líneas abiertas en la tesis realizada por el Dr. D. Leopoldo Acosta Sánchez. En aquella se hizo, entre otras cosas, una primera aproximación a las estrategias de reducción de la complejidad de la programación dinámica para sistemas lineales. Uno de los objetivos de este trabajo consiste en la formalización de dichos métodos y su extensión a sistemas no lineales. Este objetivo se ha conseguido formulando dos teoremas sobre su aplicabilidad a sistemas dinámicos en general. A pesar de la reducción obtenida, la complejidad de este algoritmo, para sistemas de orden superior al segundo, sigue siendo excesiva.

Se planteó entonces utilizar las redes neuronales, como alternativa a la programación dinámica, en la resolución de los mismos problemas de control óptimo. Se diseñó un primer método consistente en una cadena de redes neuronales que tienen como entrada el estado del sistema en una etapa y producen los comandos y el intervalo de tiempo a aplicar en dicha etapa. La ecuación dinámica del sistema, a partir de estos valores, dará el

estado en la etapa siguiente. Estas redes han de ser entrenadas en base a la función de costo elegida para producir los comandos óptimos. Al realizar su aplicación a diferentes sistemas se observó que presentaba un problema de desviación del óptimo en caso de excesivas perturbaciones.

A la vista de este comportamiento se diseñó un segundo método consistente también en una cadena de redes neuronales. Ahora, en cambio, cada una de las redes, a partir del estado actual, produce el estado en la etapa siguiente. Ha de ser la ecuación dinámica del sistema la que calcule los comandos e intervalo de tiempo para la transición indicada. Esta cadena, una vez entrenada, produce la trayectoria óptima correspondiente al índice planteado. Aunque remedia el problema del primer método, presenta dificultades para conseguir un buen entrenamiento.

Para poder comparar estas tres estrategias de control, la programación dinámica y los dos métodos de redes neuronales, hemos escogido tres sistemas no lineales de variadas características y complejidades. El primero de ellos es un sistema SISO de segundo orden que, modificando los valores de sus parámetros, puede pasar desde un sistema lineal hasta un sistema no lineal tipo *Van der Pol*. El segundo sistema es un robot móvil consistente en un vehículo de dos ruedas que puede avanzar y girar, y corresponde a un sistema MIMO de tercer orden con no linealidades trigonométricas. El tercero de los sistemas está basado en una maqueta de prácticas y consiste en dos tanques interconectados, el segundo de los cuales posee una resistencia calefactora que permite aumentar la temperatura del líquido. Su modelo es MIMO de tercer orden con parámetros distribuidos y complejidad superior a la de los anteriores.

Esta memoria comienza con dos capítulos de revisión de los principales campos tratados. En el primero se hace una breve, y necesariamente incompleta, presentación de los métodos convencionales de resolver problemas de control óptimo, a fin de situar la programación dinámica en su contexto. En la segunda parte del capítulo se hace un breve desarrollo de ésta, indicando sus conocidas alternativas y formas de aplicación. El segundo capítulo da una visión del mundo de las redes neuronales. Éste se puede también dividir en dos partes. En la primera se hace un repaso histórico de esta materia y se intenta dar una clasificación general de este campo, aún poco formalizado. En la segunda parte nos centramos en el tipo de red que será utilizada como sustituto de la programación dinámica para solucionar los problemas de control óptimo. Para ella se presentan ampliamente los dos principales algoritmos de entrenamiento.

El tercer capitulo introduce con detalle los tres sistemas ejemplo antes mencionados. Se presentará su modelo dinámico y se obtendrán las expresiones discretizadas mediante discretización clásica y funciones de bloques de pulsos.

Los tres capítulos siguientes se centran en las técnicas diseñadas y tienen estructura similar. En primer lugar explican el método correspondiente y en segundo lugar se presentan los resultados obtenidos en los sistemas ejemplos. El primero de esos capítulos trata sobre las técnicas de reducción de la complejidad de la programación dinámica. En él se formulan y demuestran teoremas de aplicación de dos de dichas técnicas. Los sistemas ejemplo sirven, además de para estudiar los resultados, para mostrar diferentes condiciones de aplicación de los teoremas. Los otros dos capítulos presentan los dos métodos basados en redes neuronales. Tras una justificación de su diseño se desarrollan las fórmulas necesarias para su entrenamiento. Posteriormente se muestran los resultados obtenidos sobre los sistemas ejemplo.

En el séptimo capítulo se analizan brevemente las herramientas informáticas utilizadas para la simulación de estos algoritmos. Se explican, a grandes rasgos, las estructuras de datos ideadas para cada uno de ellos. También se hace hincapié en la herramienta de visualización de los resultados. Tras las conclusiones de este trabajo, en el apéndice A se lista el código fuente de los principales programas realizados, tanto para simulación como para visualización de resultados. Finalmente, se presenta el listado de referencias consultadas durante la realización del presente trabajo.

Índice

Introducción xi

Índice xv

1 Control Óptimo 1

1.1 Control Óptimo de Sistemas Continuos 2

1.1.1 Tipo de Problemas 2

1.1.1.1 Problema del Control Óptimo General 2

1.1.1.2 Problema del Control Óptimo Básico 3

1.1.1.3 Problema Óptimo Lineal Cuadrático 3

1.1.2 Herramientas de Solución 4

1.1.2.1 Cálculo de Variaciones 4

1.1.2.2 Aproximación de Pontryagin 5

1.1.2.3 Aproximación de Hamilton-Jacobi 7

1.2 Sistemas Discretos 9

1.2.1 Tipo de Problemas....9
1.2.1.1 Problema de Control Óptimo Discreto....9
1.2.1.2 El Regulador Discreto Lineal Cuadrático....9
1.2.2 Herramienta de Solución....10
1.2.3 Otros Controladores....12
1.3 La Programación Dinámica....15
1.3.1 Programación Dinámica para Sistemas Discretos....15
1.3.1.1 El Regulador Discreto Lineal Cuadrático....16
1.3.2 Programación Dinámica hacia Delante....17
1.3.3 Programación Dinámica para Sistemas Continuos....18
1.3.4 Método Computacional....19
1.3.4.1 Primer Barrido....20
1.3.4.2 Segundo Barrido....21
1.3.4.3 Propiedades del Método Computacional....21
1.3.4.4 Requerimientos Computacionales....22
1.3.5 Implementación de la PD en Tiempo Real....22

2 Redes Neuronales....25
2.1 Introducción....25
2.2 Historia de las Redes Neuronales....27
2.2.1 Primeros Fundamentos....27
2.2.2 Primeras Simulaciones....28
2.2.3 Época de Recesión....29
2.2.4 Innovaciones....29
2.2.4.1 Redes Autoorganizativas....30
2.2.4.2 Memorias Asociativas.....30
2.2.4.3 Backpropagation....30
2.2.4.4 Redes Neuronales Resonantes....31
2.2.4.5 Redes Cooperativas/Competitivas....31
2.2.5 Reaparición....31
2.3 Tipos de Redes Neuronales....32
2.3.1 La Neurona Artificial....33
2.3.1.1 Función de Transferencia....33
2.3.1.2 Sesgo y Ganancias....35
2.3.1.3 Memoria....35
2.3.2 Estructura de la Red....35
2.3.3 Tipo de entrenamiento....37
2.3.3.1 Entrenamiento Supervisado....37
2.3.3.2 Entrenamiento no Supervisado....37

2.4 Arquitectura de las redes BP 38
2.5 Algoritmo de la BP 40
2.5.1 Primera Aproximación al Entrenamiento 40
2.5.2 Minimización sin Restricciones 41
2.5.3 Métodos Descendentes 42
2.5.3.1 Gradiente de una Función 42
2.5.3.2 Método Básico del Gradiente Descendente 44
2.5.3.3 Modificaciones del Método 45
2.5.4 Método del Gradiente Conjugado 46
2.5.4.1 Principios Teóricos 46
2.5.4.2 Obtención del Nuevo Algoritmo 47
2.5.4.3 Resultados del Algoritmo 49
2.5.5 Búsqueda del Paso Óptimo 51
2.5.5.1 Interpolación Cuadrática 52
2.5.6 Fórmulas de la BP 55
2.5.6.1 Problemas de la BP 57
2.6 Métodos Estadísticos de Entrenamiento 58
2.6.1 Métodos Estadísticos Básicos 58
2.6.1.1 Salto Aleatorio 58
2.6.1.2 Método del Camino Aleatorio 59
2.6.2 Método del enfriamiento progresivo 61
2.6.2.1 Introducción 61
2.6.2.2 Algoritmo de Boltzman 62
2.6.2.3 Algoritmo de Cauchy 63
2.6.2.4 Método del Calor Específico Artificial 64
2.6.2.5 Método Gradiente descendente y Cauchy 65

3 Planteamiento del Problema 67
3.1 Introducción 67
3.2 Discretización del Sistema 68
3.2.1 Discretización Clásica 68
3.2.2 Formulación BPF 69
3.3 Problemas Ejemplos 72
3.3.1 Primer Sistema 72
3.3.1.1 Ecuaciones Discretizadas 73
3.3.2 Segundo Sistema 74
3.3.2.1 Ecuaciones Discretizadas 75
3.3.3 Tercer Sistema 77
3.3.3.1 Dinámica de los Niveles 78

3.3.3.2 Dinámica de la Temperatura 79
3.3.3.3 Ecuaciones Discretizadas 81

4 Aplicación de la Programación Dinámica 83
4.1 Mejoras a la Programación Dinámica 83
4.1.1 Índices de Costo Generales 83
4.1.1.1 Propiedades de la Variable de Etapa 84
4.1.1.2 Elección de una Variable de Etapa Distinta del Tiempo 85
4.1.1.3 Formulación del Teorema 85
4.1.1.4 Proceso Para la Aplicación del Teorema 86
4.1.2 Simetría de los Sistemas 88
4.1.2.1 Proceso para la Aplicación del Teorema 89
4.1.3 Eliminación de la Interpolación 91
4.1.3.1 Modificación del Primer Barrido 91
4.1.3.2 La Discretización BPF 92
4.2 Aplicación a los Problemas Ejemplo 93
4.2.1 Primer Sistema 93
4.2.1.1 Elección de la Variable de Etapa 93
4.2.1.2 Propiedad de Simetría 94
4.2.1.3 Expresiones para el Comando y el Tiempo 95
4.2.1.4 Resultados Obtenidos 95
4.2.2 Segundo Sistema 97
4.2.2.1 Elección de la Variable de Etapa 98
4.2.2.2 Propiedad de Simetría 100
4.2.2.3 Expresiones para el Comando y el Tiempo 100
4.2.2.4 Resultados Obtenidos 100
4.2.3 Tercer Sistema 102
4.2.3.1 Elección de la Variable de Etapa 102
4.2.3.2 Propiedad de Simetría 104
4.2.3.3 Expresiones del Comando y el Tiempo 104
4.2.3.4 Resultados Obtenidos 104
4.2.4 Resumen y Conclusiones 106

5 Primer Método Basado en Redes Neuronales 107
5.1 RN como Alternativa a la PD 107
5.2 Descripción del Método 109
5.2.1 Cálculo de las Trayectorias 111
5.2.2 Proceso de Entrenamiento 111

5.2.3 Fórmulas para la Backpropagation 112
5.2.4 Estudio de la Complejidad 115
5.3 Aplicación a los Problemas Ejemplo 116
5.3.1 Primer Sistema 116
5.3.1.1 Expresiones Necesarias 116
5.3.1.2 Resultados Obtenidos 118
5.3.2 Segundo Sistema 120
5.3.2.1 Expresiones Necesarias 120
5.3.2.2 Resultados 121
5.3.3 Tercer Sistema 123
5.3.3.1 Expresiones Necesarias 123
5.3.3.2 Resultados 125
5.4 Resumen y Conclusiones 127

6 Segundo Método Basado en Redes Neuronales 129
6.1 Descripción del Método 129
6.1.1 Cálculo de las Trayectorias 131
6.1.2 Proceso de Entrenamiento 131
6.1.3 Modificación de la Función de Costo 135
6.1.4 Problemas del Entrenamiento 137
6.1.4.1 Estancamiento en Mínimos Locales 138
6.1.4.2 Falta de Sensibilidad 139
6.1.5 Ayudas al Entrenamiento 139
6.1.5.1 Entrenamiento Individual 139
6.1.5.2 Trayectoria Deseada 140
6.1.5.3 Zonas Prohibidas 141
6.1.5.4 Gradiente Escalado 141
6.2 Aplicación a los Problemas Ejemplo 142
6.2.1 Primer Sistema 142
6.2.1.1 Expresiones Necesarias 142
6.2.1.2 Resultados Obtenidos 145
6.2.2 Segundo Sistema 147
6.2.2.1 Expresiones Necesarias 147
6.2.2.2 Resultados 152
6.2.3 Tercer Sistema 154
6.2.3.1 Expresiones Necesarias 154
6.2.3.2 Resultados 157
6.2.4 Resumen y Conclusiones 158

7 Implementación 159
7.1 Características Generales 159
7.1.1 Sistema Operativo 159
7.1.2 Lenguaje de Programación 160
7.1.3 Estudio de los Resultados 161
7.2 Codificación 161
7.2.1 Programación Dinámica 162
7.2.1.1 Estructura de Datos 162
7.2.1.2 Algoritmos 162
7.2.2 Métodos de Redes Neuronales 164
7.2.2.1 Estructura de Datos 164
7.2.2.2 Algoritmos 165
7.2.3 Estudio en MATLAB 167
7.2.3.1 Estructuras de Datos 167
7.2.3.2 Nuevas Funciones 169

Principales Aportaciones, Conclusiones y Lineas Abiertas 171

Apéndice A 175
A.1 Programación Dinámica 175
A.1.1 Primer Sistema 176
A.1.2 Segundo Sistema 187
A.1.3 Tercer Sistema 195
A.2 Redes Neuronales 204
A.2.1 Código Común 204
A.2.2 Primer Sistema 231
A.2.3 Segundo Sistema 236
A.2.4 Tercer Sistema 241
A.3 Tratamiento de Resultados 248
A.3.1 Manejo de Trayectorias 248
A.3.2 Representación Gráfica 254
A.3.3 Otros 263

Referencias 267

1

Control Óptimo

En este primer capítulo se realiza un repaso general de las técnicas clásicas de solución de los problemas de control óptimo. Nos limitaremos a los problemas deterministas ya que son sólo estos los que han sido tratados en el resto del trabajo. Por lo conocido del tema, haremos una presentación esquemática en la que destacaremos los casos y dificultades de aplicación de los distintos métodos. No pretendemos, por lo tanto, hacer una exposición exhaustiva ni rigurosa sino sencillamente dar una visión global de los métodos más utilizados en el control óptimo.

Dividiremos estas técnicas en dos grupos: las diseñadas para sistemas continuos y las que abordan sistemas discretos, y las trataremos en las dos primeras secciones del capítulo. En la última sección nos extenderemos algo más en la presentación de la programación dinámica ya que, aunque también conocida, es uno de los métodos de solución estudiados en este trabajo.

1.1 Control Óptimo de Sistemas Continuos

Presentaremos en primer lugar los tipos en que se pueden clasificar los problemas de control óptimo continuo. En segundo lugar presentaremos las técnicas de solución y que posibilidad y dificultad de aplicación tienen a los problemas mencionados [Shul 67] [Pun 69] [Luen 79] [Kail 80] [Barn 85] [Rubi 86] [Sont 90].

1.1.1 Tipo de Problemas

Dividiremos los problemas de control óptimo en tres tipos principales. Comenzaremos por el más general y seguiremos con los más restrictivos.

1.1.1.1 Problema del Control Óptimo General

El problema del control óptimo general (PCOG) es el problema más generico con un significado práctico. Su planteamiento es el siguiente:

Sea la planta descrita por la ecuación diferencial:

$$\dot{\underline{x}} = \underline{f}(\underline{x},\underline{u},t) \tag{1.1}$$

donde $\underline{x}\in\mathsf{R}^n$ y $\underline{u}\in\mathsf{R}^m$. El rendimiento de este sistema es juzgado por una integral del índice de calidad de la forma

$$J = \int_{t_i}^{t_f} L(\underline{x},\underline{u},t)dt + S\left[\underline{x}(t_f),t_f\right] \tag{1.2}$$

donde las funciones L y S son definidas positivas de sus argumentos. Es posible transformar este índice de calidad en un índice de tipo integral de la siguiente manera

$$J = \int_{t_i}^{t_f}\left[L(\underline{x},\underline{u},t) + \frac{d}{dt}S(\underline{x},t)\right]dt \tag{1.3}$$

Las condiciones de contorno son las siguientes:

- El tiempo final t_f puede ser fijo o libre.
- El estado final $\underline{x}_f$ puede estar fijo, completamente libre, o especificado por un conjunto de relaciones.
- Se requiere que el vector de control sea miembro de un conjunto U denominado región de control. Dicha región puede ser abierta o cerrada y limitada o ilimitada.

El requerimiento de que $\underline{u}$ debe pertenecer a la región de control U es necesario en el caso práctico de que los sistemas no dispongan de un control ilimitado.

1.1.1.2 Problema del Control Óptimo Básico

El problema del control óptimo básico (PCOB) es un subconjunto del PCOG. Al igual que en aquél se asume que la planta a controlar es general y se describe mediante una ecuación diferencial de la forma

$$\dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u}, t) \tag{1.4}$$

donde $\underline{x} \in \mathrm{R}^n$ y $\underline{u} \in \mathrm{R}^m$. El rendimiento de este sistema es juzgado por una integral del índice de calidad de la forma

$$J = \int_{t_i}^{t_f} L(\underline{x}, \underline{u}, t)dt \tag{1.5}$$

que en este caso se plantea integral puro.

Las condiciones de contorno son las siguientes:

- El tiempo inicial t_i y final t_f son fijos.
- El estado final $\underline{x}_f$ puede estar fijo, completamente libre, o especificado por un conjunto de relaciones.
- El vector de control $\underline{u}$ se supone ilimitado.

Las diferencias con el caso anterior es que el intervalo de tiempo está fijado y, por otro lado, se elimina la restricción del comando.

1.1.1.3 Problema Óptimo Lineal Cuadrático

El problema óptimo lineal cuadrático (POLC) es un subconjunto del PCOB para el cual el sistema está descrito por una ecuación de estado lineal

$$\dot{\underline{x}} = \mathbf{A}\underline{x} + \mathbf{B}\underline{u} \tag{1.6}$$

y el índice de calidad es cuadrático

$$J = \int_0^{\infty} (\underline{x}^{\mathrm{T}}\mathbf{Q}\underline{x} + \underline{u}^{\mathrm{T}}\mathbf{R}\underline{u})dt \tag{1.7}$$

donde **Q** es simétrica y semidefinida positiva, y **R** es también simétrica y definida positiva. Como en el PCOB se asume que $\underline{u}$ está ilimitada. También se puede plantear este problema para un periodo de tiempo entre *0* y *T* finito.

La diferencia con el PCOB es que el sistema ha de ser lineal y la función de costo cuadrática. Aunque se trata de un problema restrictivo, muchos problemas complejos se pueden aproximar a uno de este tipo que, como veremos, posee solución sencilla.

1.1.2 Herramientas de Solución

Una vez planteado un problema de control óptimo, la solución consiste en encontrar el control óptimo $\underline{u}^o(t)$ o la ley de control óptimo $\underline{u}^o=\underline{u}^o[\underline{x}(t),t]$ que transfiere al sistema, desde cierto estado o condición inicial $\underline{x}_i=\underline{x}(t_i)$ hasta cierto estado o condición final $\underline{x}_f=\underline{x}(t_f)$, minimizando la integral del índice de calidad. La trayectoria correspondiente al control óptimo o a la ley de control óptima es denominada trayectoria óptima.

Existen varias herramientas para obtener dicha solución. Las presentaremos e indicaremos a que problemas y bajo que condiciones se puede aplicar.

1.1.2.1 Cálculo de Variaciones

Del calculo de variaciones sabemos que dado el variacional:

$$V(\underline{x})=\int_{t_i}^{t_f} L\left[\underline{x}(t),\dot{\underline{x}}(t),t\right]dt \tag{1.8}$$

donde $\underline{x}\in\mathrm{R}^n$, las funciones $\underline{x}(t)$ que lo minimizan han de ser extremales. Se denominan extremales a aquellas funciones que satisfacen la ecuación de Euler

$$\frac{\partial L(\underline{x},\dot{\underline{x}},t)}{\partial x_i}-\frac{d}{dt}\frac{\partial L(\underline{x},\dot{\underline{x}},t)}{\partial \dot{x}_i}=0 \qquad i=1,2,\ldots,n \tag{1.9}$$

Para determinar completamente las soluciones de esta ecuación son precisas $2n$ condiciones de contorno. Si el punto inicial y final están fijados, es decir,

$$\underline{x}(t_i)=\underline{x}^i \qquad \text{y} \qquad \underline{x}(t_f)=\underline{x}^f \tag{1.10}$$

con estas condiciones se determina la solución. En caso de ser alguno de los extremos libres, se debe aplicar la condición de contorno generalizada. Para el punto final ésta es

$$\delta\underline{x}^{\mathrm{T}}\frac{\partial L(\underline{x},\dot{\underline{x}},t)}{\partial \dot{\underline{x}}}\bigg|_{t_f}+\left[L(\underline{x},\dot{\underline{x}},t)-\dot{\underline{x}}^{\mathrm{T}}\frac{\partial L(\underline{x},\dot{\underline{x}},t)}{\partial \dot{\underline{x}}}\right]\bigg|_{t_f}\delta t_f=0 \tag{1.11}$$

Si el punto final es completamente libre, al ser $\delta\underline{x}^{\mathrm{T}}$ y δt_f independientes, se ha de cumplir que

$$\frac{\partial L(\underline{x},\dot{\underline{x}},t)}{\partial x_i}\bigg|_{t_f}=0 \qquad i=1,2,\ldots,n \tag{1.12}$$

$$\left[L(\underline{x},\dot{\underline{x}},t)-\dot{\underline{x}}^{\mathrm{T}}\frac{\partial L(\underline{x},\dot{\underline{x}},t)}{\partial \dot{\underline{x}}}\right]\bigg|_{t_f}=0 \tag{1.13}$$

En caso de que el punto final deba cumplir cierta condición, denominada condición transversal, ésta se ha de traducir en una relación entre $\delta\underline{x}^{\mathrm{T}}$ y δt_f. Las condiciones de contorno serán la ecuación (1.11) junto con la condición transversal.

De forma similar se puede definir la condición de contorno generalizada para el punto inicial.

Este método es sólo aplicable a un PCOB ya que en el planteamiento variacional las variables no están limitadas. Se ha de despejar de las ecuaciones de la planta el vector de control, es decir, se ha de obtener $\underline{u}=\underline{g}(\underline{x},\dot{\underline{x}},t)$. En ese caso la función de costo se reescribirá como

$$J=\int_{t_f}^{t_i} L\left[\underline{x},\underline{g}(\underline{x},\dot{\underline{x}},t),t\right]dt=\int_{t_f}^{t_i} L'(\underline{x},\dot{\underline{x}},t)dt \tag{1.14}$$

la cual presenta la forma del planteamiento variacional. Una vez encontrado $\underline{x}(t)$, a través de las ecuaciones de Euler y las condiciones de contorno, se podrá obtener $\dot{\underline{x}}(t)$ y con ella calcular $\underline{u}(t)$ mediante $\underline{g}(\underline{x},\dot{\underline{x}},t)$.

Aunque esta solución, en teoría, puede ser aplicada a cualquier problema, la aplicación en la práctica es bastante difícil. Esto es debido al hecho de que en raras ocasiones se puede despejar, de forma sencilla, $\underline{u}$ en términos de $\underline{x}$, $\dot{\underline{x}}$ y t.

1.1.2.2 Aproximación de Pontryagin

Para aplicar este método se define la función de estado de Pontryagin. Ésta tiene la forma siguiente

$$H(\underline{x},\underline{u},\underline{\lambda},t)=L(\underline{x},\underline{u},t)+\underline{\lambda}^{\mathrm{T}}(t)\,\underline{f}(\underline{x},\underline{u},t) \tag{1.15}$$

donde $\underline{\lambda}(t)$ corresponde a las funciones multiplicadoras de Lagrange. Con esta definición la condición de que la solución sea extremal y cumpla la ecuación dinámica del sistema se traduce en el cumplimiento de las siguientes ecuaciones

$$\dot{\underline{\lambda}}=-\frac{\partial H(\underline{x},\underline{u},\underline{\lambda},t)}{\partial \underline{x}} \tag{1.16}$$

$$\frac{\partial H(\underline{x},\underline{u},\underline{\lambda},t)}{\partial \underline{u}}=0 \tag{1.17}$$

$$\dot{\underline{x}}=\frac{\partial H(\underline{x},\underline{u},\underline{\lambda},t)}{\partial \underline{\lambda}} \tag{1.18}$$

Estas tres ecuaciones han de ser resueltas simultáneamente sujetas a las condiciones de contorno. Esto generalmente se realiza resolviendo primero la ecuación algebraica (1.17) obteniendo así el control óptimo $\underline{u}^o$ en términos de $\underline{x}$, $\underline{\lambda}$ y t

$$\underline{u}^o = \underline{u}^o(\underline{x},\underline{\lambda},t) \tag{1.19}$$

Este resultado es entonces sustituido en la función H para crear la función de estado óptima de Pontryagin $H^o(\underline{x},\underline{\lambda},t)=H[\underline{x},\underline{u}^o(\underline{x},\underline{\lambda},t),\underline{\lambda},t]$. Las ecuaciones diferenciales (1.16) y (1.18) se convierten en un conjunto de $2n$ ecuaciones diferenciales de primer orden acopladas:

$$\dot{\underline{\lambda}} = -\frac{\partial H^o(\underline{x},\underline{u},\underline{\lambda},t)}{\partial \underline{x}} \tag{1.20}$$

$$\dot{\underline{x}} = \frac{\partial H^o(\underline{x},\underline{u},\underline{\lambda},t)}{\partial \underline{\lambda}} \tag{1.21}$$

La trayectoria óptima $\underline{x}^o(t)$ y $\underline{\lambda}(t)$ se encuentran resolviendo estas ecuaciones con las $2n$ condiciones de contorno. En caso de punto final libre se ha de plantear la condición de contorno generalizada para el punto final

$$\left[\frac{\partial S(\underline{x},t)}{\partial \underline{x}} - \underline{\lambda}\right]^{\mathrm{T}} d\underline{x}\Big|_{t_f} + \left[H^o(\underline{x},\underline{u},t) + \frac{\partial S(\underline{x},t)}{\partial t}\right] dt\Big|_{t_f} = 0 \tag{1.22}$$

Con este planteamiento es posible aplicarlo al PCOB ya que $\underline{u}$ no ha de estar limitada. Para poder aplicar este desarrollo al PCOG debemos poder tratar con comandos limitados. Para ello hacemos uso del principio de mínimo de Pontryagin. Éste nos dice que el control óptimo $\underline{u}^o(\underline{x},\underline{\lambda},t)$ se obtiene minimizando $H(\underline{x},\underline{u},\underline{\lambda},t)$ con respecto a los controles $\underline{u}$ en la región de control dada, mientras se trata al resto de variables como constantes. En otras palabras $\underline{u}^o(\underline{x},\underline{\lambda},t)$ es el vector de control admisible para el cual $H(\underline{x},\underline{u},\underline{\lambda},t)$ tiene su mínimo valor. La función de estado óptima de Pontryagin queda entonces

$$H^o(\underline{x},\underline{\lambda},t) = H\left[\underline{x},u^o(\underline{x},\underline{\lambda},t),\underline{\lambda},t\right] = \min_{u\in U} H(\underline{x},\underline{u},\underline{\lambda},t) \tag{1.23}$$

A pesar de la generalidad de este método la solución real de las ecuaciones (1.20) y (1.21), y en su caso (1.22), presenta varios problemas prácticos. El primero es que las ecuaciones, como suelen ser no lineales y variables en el tiempo, sólo pueden resolverse numéricamente. El segundo es el de la mezcla de las condiciones de contorno: n condiciones de contorno están dadas en el tiempo inicial y las otras n en el tiempo final. Además, una vez resuelto el problema, tendremos un control óptimo en lazo abierto $\underline{u}^o(t)$. Con todo esto, existen aplicaciones para sistemas e índices de costo sencillos como son los controles Bang-Bang para sistemas de segundo y tercer orden con índices de tiempo mínimo o tiempo-comando [Atha 66] [Ryan 82] [Luqu 83].

1.1.2.3 Aproximación de Hamilton-Jacobi

Para este desarrollo se supone que existe una ley de control óptimo $\underline{u}^o(\underline{x},t)$ donde la dependencia con el tiempo es debida a la variación temporal inherente al problema y no debido a la dependencia con el estado inicial.

Se define la función escalar $V(\underline{x},t)$ como el mínimo valor del índice de calidad para un estado inicial $\underline{x}$ en el tiempo t, es decir,

$$V(\underline{x},t) = \int_t^{t_f} L\left[x(\tau), u^o(\underline{x},\tau), \tau\right] d\tau \tag{1.24}$$

Si sustituimos $\underline{\lambda}$ por $\underline{\nabla}V$ en la ecuación de estado de Pontryagin tendremos

$$H(\underline{x},\underline{u},\underline{\nabla}V,t) = L(\underline{x},\underline{u},t) + \underline{\nabla}V^{\mathrm{T}}\,\underline{f}(\underline{x},\underline{u},t) \tag{1.25}$$

En primer lugar se ha de determinar $\underline{u}^o=\underline{u}^o(\underline{x},\underline{\nabla}V,t)$ resolviendo la ecuación algebraica

$$\frac{\partial H(\underline{x},\underline{u},\underline{\nabla}V,t)}{\partial \underline{u}} = 0 \tag{1.26}$$

sustituyendo este resultado en H tendremos $H^o(\underline{x},\underline{\nabla}V,t)$. Para obtener $V(\underline{x},t)$ se debe resolver la ecuación de Hamilton-Jacobi:

$$H^o\left[\underline{x},\underline{\nabla}V(\underline{x},t),t\right] + \frac{\partial V(\underline{x},t)}{\partial t} = 0 \tag{1.27}$$

Para poder determinar $V(\underline{x},t)$ completamente es necesario especificar las condiciones de contorno apropiadas. En el caso de estado final fijo, es necesario que $V(\underline{x},t)$ sea cero para $\underline{x}=x_f$ y $t=t_f$ y distinto de cero para el resto. Si, en cambio, el estado final es libre, $V(\underline{x},t_f)$ debe ser cero para cualquier $\underline{x}$.

Una vez que se obtiene $V(\underline{x},t)$ se puede hallar su gradiente y sustituirlo en la expresión $\underline{u}^o(\underline{x},\underline{\nabla}V,t)$ para crear la ley de control óptimo.

Este método puede aplicarse teóricamente tanto a PCOB como a PCOG pero, desafortunadamente, tiene dos dificultades que limitan seriamente su utilidad. En primer lugar, es prácticamente imposible resolver la ecuación de Hamilton-Jacobi salvo para problemas triviales, además de que no se conoce una técnica general de solución. En segundo lugar, aunque se resuelva esta ecuación, la ley de control suele ser impracticable. Por esta razón la ecuación de Hamilton-Jacobi no se acepta, y es necesario resolver la aproximación de Pontryagin.

En favor de la aproximación de Hamilton-Jacobi está el hecho de que para el POLC obtiene una solución sencilla. En este caso la función H queda

$$H(\underline{x},\underline{u},\underline{\nabla}V,t) = \underline{\nabla}V^{\mathrm{T}}(\mathbf{A}\underline{x} + \mathbf{B}\underline{u}) + \underline{x}^{\mathrm{T}}\mathbf{Q}\underline{x} + \underline{u}^{\mathrm{T}}\mathbf{R}\underline{u} \tag{1.28}$$

haciendo $\partial H/\partial u=0$ obtenemos que $\underline{u}^o(\underline{x},t)$ viene dado por

$$\underline{u}^o = -\tfrac{1}{2}\mathbf{R}^{-1}\mathbf{B}^{\mathrm{T}}\underline{\nabla}V(\underline{x},t) \tag{1.29}$$

Sustituyendo este resultado en H, la ecuación de Hamilton-Jacobi, para tiempo final finito, se resuelve asumiendo que la solución es una forma cuadrática variante con el tiempo, es decir

$$V(\underline{x},t) = \underline{x}^{\mathrm{T}}\mathbf{P}(t)\underline{x} = \int_t^T (\underline{x}^{\mathrm{T}}\mathbf{Q}\underline{x} + \underline{u}^{\mathrm{T}}\mathbf{R}\underline{u})dt \tag{1.30}$$

donde $\mathbf{P}(t)$ es una matriz simétrica definida positiva para todo $t<T$ y , por la condición de contorno, $\mathbf{P}(T)=0$. Para que se satisfaga la ecuación de Hamilton-Jacobi, es necesario que $\mathbf{P}(t)$ satisfaga la siguiente ecuación diferencial

$$\dot{\mathbf{P}}(t) + \mathbf{Q} - \mathbf{P}(t)\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{P}(t) + \mathbf{P}(t)\mathbf{A} + \mathbf{A}^{\mathrm{T}}\mathbf{P}(t) = 0 \tag{1.31}$$

sujeta a la condición de contorno $\mathbf{P}(T)=0$. Esta ecuación es conocida como la ecuación de Riccati matricial.

Aunque se dispone de métodos generales para obtener una solución analítica de $\mathbf{P}(t)$, estos son inmanejables para sistemas de orden superior al segundo. Por otro lado es muy sencillo obtener una solución numérica en una computadora digital. Esto se hace integrando la ecuación de Riccati hacia atrás sobre el intervalo de tiempo de interés, desde la condición terminal conocida.

Una vez se conoce $\mathbf{P}(t)$ para $0\le t\le T$, la ley de control óptimo se obtiene de la ecuación (1.29) utilizando que $\underline{\nabla}V=2\mathbf{P}(t)\underline{x}$

$$\underline{u}^o(\underline{x},t) = \mathbf{R}^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{P}(t)\underline{x} \tag{1.32}$$

que se suele escribir de la forma

$$\underline{u}^o(\underline{x},t) = -\mathbf{K}(t)^{\mathrm{T}}\underline{x} \tag{1.33}$$

donde

$$\mathbf{K}(t) = \mathbf{R}(t)\mathbf{B}\mathbf{P}^{-1} \tag{1.34}$$

En el caso en que $T=\infty$ la matriz $\mathbf{P}(t)$ se convierte en constante que suele designarse por $\mathbf{P}_0$. Esta matriz se puede encontrar de diversas maneras. Una manera es calcular el límite de la solución $\mathbf{P}(t)$ cuando $t\to\infty$. En la práctica esta técnica no es utilizada porque usualmente no se conoce la expresión analítica de $\mathbf{P}(t)$. Lo que se suele hacer es integrar la ecuación de Riccati hacia atrás en el tiempo, desde la conocida condición final $\mathbf{P}(\infty)=0$, hasta que se obtenga el grado de exactitud deseado. También puede encontrarse utilizando el hecho de que es constante y, por ello, la ecuación de Riccati queda

$$\mathbf{Q} - \mathbf{P}_0\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^{\mathrm{T}}\mathbf{P}_0 + \mathbf{P}_0\mathbf{A} + \mathbf{A}^{\mathrm{T}}\mathbf{P}_0 = 0 \tag{1.35}$$

la cual se denomina ecuación de Riccati reducida o degenerada. En este caso tenemos que resolver *n(n+1)/2* ecuaciones algebraicas no lineales. De las distintas posibilidades la respuesta deseada se obtiene exigiendo que $\mathbf{P}_0$ sea definida positiva.

1.2 Sistemas Discretos

Vamos a proceder de forma similar a como hicimos en el caso continuo, presentando en primer lugar la formulación de los problemas y posteriormente las herramientas de solución [Åstr 84] [Iser 89] [Fran 90].

1.2.1 Tipo de Problemas

1.2.1.1 Problema de Control Óptimo Discreto

El problema del control óptimo discreto (PCOD) se plantea de la siguiente manera:

Sea una planta descrita por una ecuación dinámica no lineal de tiempo discreto general

$$\underline{x}_{k+1} = \underline{f}(\underline{x}_k, \underline{u}_k, k) \tag{1.36}$$

Supongamos un índice de calidad escalar general de la forma

$$J = S(N, \underline{x}_N) + \sum_{k=i}^{N-1} L(\underline{x}_k, \underline{u}_k, k) \tag{1.37}$$

donde el intervalo de tiempo que nos interesa es [*i,N*]. Este intervalo, así como los estados iniciales y finales, pueden ser completamente libres.

1.2.1.2 El Regulador Discreto Lineal Cuadrático

El Regulador Discreto Lineal Cuadrático (RDLC) como ocurría en el caso continuo es un caso particular del PCOD que tiene interés porque se le encontrará solución sencilla y es posible aproximar problemas más complejos a uno de este tipo.

En este caso la planta se describe por la ecuación lineal discreta

$$\underline{x}_{k+1} = \mathbf{A}_k \underline{x}_k + \mathbf{B}_k \underline{u}_k \tag{1.38}$$

con $\underline{x}_k \in \mathrm{R}^n$ y $\underline{u}_k \in \mathrm{R}^m$. El índice de calidad asociado es la función cuadrática

$$J = \frac{1}{2} \underline{x}_N^{\mathrm{T}} \mathbf{S}_N \underline{x}_N + \frac{1}{2} \sum_{k=i}^{N-1} (\underline{x}_k^{\mathrm{T}} \mathbf{Q}_k \underline{x}_k + \underline{u}_k^{\mathrm{T}} \mathbf{R}_k \underline{u}_k) \tag{1.39}$$

definida sobre el intervalo de tiempo de interés [*i,N*]. Tanto la planta como las matrices de costo pueden, en general, variar con el tiempo. El estado inicial de la planta viene dado por $\underline{x}_i$. Asumimos que $\mathbf{Q}_k$ y $\mathbf{S}_N$ son simétricas y semidefinidas positivas; y $\mathbf{R}_k$ es simétrica y definida positiva. Como en el caso continuo se supone ilimitado el control $\underline{u}_k$.

1.2.2 Herramienta de Solución

El objetivo es encontrar el control $\underline{u}_k^o$ en el intervalo $[i,N]$ que conduce al sistema a través de la trayectoria $\underline{x}_k^o$ tal que se minimice el índice de calidad. En este caso, la solución se obtiene aplicando los multiplicadores de Lagrange al problema de minimización del costo J, para poder así incluir las restricciones que suponen las ecuaciones del sistema. Como las restricciones están especificadas para cada tiempo k, se definirán multiplicadores de Lagrange para cada instante $\underline{\lambda}_k \in R^n$. Definiendo la función Hamiltoniana como

$$H(\underline{x}_k, \underline{u}_k, k) = L(\underline{x}_k, \underline{u}_k, k) + \underline{\lambda}_{k+1}^{\mathrm{T}} f(\underline{x}_k, \underline{u}_k, k) \tag{1.40}$$

las condiciones necesarias para el mínimo vienen dadas por

$$\underline{x}_{k+1} = \frac{\partial H_k}{\partial \underline{\lambda}_{k+1}} \qquad k = i, \ldots, N-1 \tag{1.41}$$

$$\underline{\lambda}_k = \frac{\partial H_k}{\partial \underline{x}_k} \qquad k = i, \ldots, N-1 \tag{1.42}$$

$$\frac{\partial H_k}{\partial \underline{u}_k} = 0 \qquad k = i, \ldots, N-1 \tag{1.43}$$

y las condiciones de contorno necesarias para resolver las anteriores

$$\left(\frac{\partial S}{\partial \underline{x}_N} - \underline{\lambda}_N \right) d\underline{x}_N = 0 \tag{1.44}$$

$$\left(\frac{\partial H_i}{\partial \underline{x}_i} \right)^{\mathrm{T}} d\underline{x}_i = 0 \tag{1.45}$$

donde la primera se da sólo en el tiempo final $k=N$ y la segunda se da para el tiempo inicial $k=i$. Si el estado inicial está fijo entonces $d\underline{x}_i=0$, por lo que (1.45) es cierta independiente del valor de H. Por otro lado, en el caso de estado inicial libre es necesario que

$$\frac{\partial H_i}{\partial \underline{x}_i} = 0 \tag{1.46}$$

Para el estado final, en el caso de que sea fijo, usaremos el valor de $\underline{x}_N$ como condición terminal. Si $\underline{x}_N$ puede variar al determinar la solución óptima, se requiere que

$$\underline{\lambda}_N = \frac{\partial S}{\partial \underline{x}_N} \tag{1.47}$$

lo que supone la condición final.

Estas son expresiones generales para el PCOD, pero es muy difícil deducir expresiones explícitas. Un caso especial importante son los sistemas lineales con índices de calidad cuadráticos para los cuales, como en el caso continuo, se encuentran soluciones analíticas.

Para resolver el problema del regulador discreto lineal cuadrático (RDLC), comenzamos definiendo la función Hamiltoniana

$$H_k = \frac{1}{2}(\underline{x}_k^{\mathrm{T}}\mathbf{Q}_k\,\underline{x}_k + \underline{u}_k^{\mathrm{T}}\mathbf{R}_k\,\underline{u}_k) + \underline{\lambda}_{k+1}^{\mathrm{T}}(\mathbf{A}_k\,\underline{x}_k + \mathbf{B}_k\,\underline{u}_k) \tag{1.48}$$

Las ecuaciones de la (1.41) a la (1.43) quedan entonces

$$\underline{x}_{k+1} = \frac{\partial H_k}{\partial \underline{\lambda}_{k+1}} = \mathbf{A}_k\,\underline{x}_k + \mathbf{B}_k\,\underline{u}_k \tag{1.49}$$

$$\underline{\lambda}_k = \frac{\partial H_k}{\partial \underline{x}_k} = \mathbf{Q}_k\,\underline{x}_k + \mathbf{A}_k^{\mathrm{T}}\,\underline{\lambda}_{k+1} \tag{1.50}$$

$$0 = \frac{\partial H_k}{\partial \underline{u}_k} = \mathbf{R}_k\,\underline{u}_k + \mathbf{B}_k^{\mathrm{T}}\,\underline{\lambda}_{k+1} \tag{1.51}$$

con lo que

$$\underline{u}_k = -\mathbf{R}_k^{-1}\mathbf{B}_k^{\mathrm{T}}\,\underline{\lambda}_{k+1} \tag{1.52}$$

La secuencia de control óptimo estará determinada si podemos encontrar la secuencia $\underline{\lambda}_k$. Para hacer esto debemos usar las condiciones de contorno. Como habitualmente conocemos el estado inicial, para la condición final se presentan dos casos:

- **Punto final fijo.** Nuestro objetivo será hacer que $\underline{x}_N$ concuerde exactamente con el estado final de referencia $\underline{r}_N$. Por esta igualdad la contribución del estado final a *J* tendrá siempre un valor fijo, y será redundante incluirlo. Por ello hacemos $\mathbf{S}_N=0$. Para poder alcanzar una solución útil debemos hacer también $\mathbf{Q}=0$, con lo que estamos pidiendo unos controles de mínima energía. Las ecuaciones a resolver están desacopladas de la ecuación de estado, por lo que el problema tiene fácil solución. Operando tenemos que

$$\lambda_N = -\mathbf{G}_{0,N}^{-1}(\underline{r}_N - \mathbf{A}^N\,\underline{x}_0) \tag{1.53}$$

donde

$$\mathbf{G}_{0,N} = \sum_{j=0}^{N-1}\mathbf{A}^{N-j-1}\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^{\mathrm{T}}(\mathbf{A}^{\mathrm{T}})^{N-j-1} \tag{1.54}$$

y por lo tanto la secuencia de control óptimo es

$$\underline{u}_k^o = \mathbf{R}^{-1}\mathbf{B}^{\mathrm{T}}(\mathbf{A}^{\mathrm{T}})^{N-k-1}\mathbf{G}_{0,N}^{-1}(\underline{r}_N - \mathbf{A}^N \underline{x}_0) \tag{1.55}$$

que es la solución de control de mínima energía al problema del RDLC. Este control óptimo es en lazo abierto, puede ser precalculado conociendo sólo el estado inicial y el deseado, y es independiente de los estados intermedios que tome el sistema durante el intervalo [*0,N*].

- **Estado final libre**. Se ha de cumplir la ecuación (1.47) que en este caso, por la forma de la función *S*, implica

$$\underline{\lambda}_N = \mathbf{S}_N \underline{x}_N \tag{1.56}$$

que representa la nueva condición terminal. Para resolver estas ecuaciones asumiremos que esta última relación lineal se mantiene para todos los instantes $k \leq N$

$$\underline{\lambda}_k = \mathbf{S}_k \underline{x}_k \tag{1.57}$$

Donde $\mathbf{S}_k$ se obtiene mediante la ecuación de Riccati

$$\mathbf{S}_k = \mathbf{A}_k^{\mathrm{T}}\left[\mathbf{S}_{k+1} - \mathbf{S}_{k+1}\mathbf{B}_k(\mathbf{B}_k^{\mathrm{T}}\mathbf{S}_{k+1}\mathbf{B}_k + \mathbf{R}_k)^{-1}\mathbf{B}_k^{\mathrm{T}}\mathbf{S}_{k+1}\right]\mathbf{A}_k + \mathbf{Q}_k \tag{1.58}$$

la cual representa una recursión hacia atrás sobre las matrices $\mathbf{S}_k$ donde el caso base es la matriz $\mathbf{S}_N$. La secuencia intermedia $\mathbf{S}_k$ puede ser calculada previamente, conociendo sólo la planta y los parámetros del índice de calidad. Definiendo la secuencia de ganancia de Kalman como

$$\mathbf{K}_k = (\mathbf{B}_k^{\mathrm{T}}\mathbf{S}_{k+1}\mathbf{B}_k + \mathbf{R}_k)^{-1}\mathbf{B}_k^{\mathrm{T}}\mathbf{S}_{k+1}\mathbf{A}_k \tag{1.59}$$

tenemos que

$$\underline{u}_k = -\mathbf{K}_k \underline{x}_k \tag{1.60}$$

La ganancia de Kalman puede ser calculada antes de que el control sea aplicado a la planta. Es una realimentación de variables de estado dependientes del tiempo, la cual expresa el control necesario en términos del estado actual, dando con ello una ley de control en lazo cerrado. Si las matrices del sistema y el índice de calidad son invariantes con el tiempo, la ganancia de realimentación $\mathbf{K}_k$ es todavía una función del tiempo, ya que, en general, la solución de la ecuación de Riccati $\mathbf{S}_k$ es dependiente del tiempo. Esta variabilidad con la etapa hace incómoda su utilización ya que es necesario almacenar la secuencia completa de matrices $\mathbf{K}_k$. Por ello, en algunas implementaciones se opta por una realimentación subóptima tomando una matriz $\mathbf{K}$ constante.

1.2.3 Otros Controladores

La formulación del control óptimo para plantas polinomiales hace que aparezcan otros controladores que en la actualidad gozan de especial aceptación. En esta línea los desarrollos más recientes [Cama 95] [Pére 94] tienen su origen en el controlador predictivo generalizado (GPC) que introdujo Clarke en 1987 [Clar 87] [Clar 89].

Limitándonos a un plantemiento determinista para un sistema SISO, consideremos el modelo siguiente para la planta

$$A(z^{-1})\Delta y(t) = B(z^{-1})\Delta u(t-1) \tag{1.61}$$

donde $\Delta=1-z^{-1}$, $y(t)$ es la salida y $u(t)$ es la entrada de control. Los polinomios $A(z^{-1})$ y $B(z^{-1})$ se expresan en términos del operador retardo z^{-1}, con órdenes n_a y n_b respectivamente.

El GPC esta basado en el concepto de predicción de largo rango y con *receding horizon*. Es decir, la predicción se realiza j etapas hacia adelante, obteniendo una secuencia de comandos asociados a cada etapa. Sin embargo de toda esta secuencia sólo el primer comando es aplicado en la etapa actual. Este procedimiento se repite en cada instante de muestreo.

El predictor óptimo, conocida la salida hasta el instante t y las entradas a aplicar al sistema en el futuro, se encuentra que es :

$$\hat{y}(t+j/t) = G_j(z^{-1})\Delta u(t+j-1) + F_j(z^{-1})y(t) \tag{1.62}$$

donde

$$G_j(z^{-1}) = E_j(z^{-1})B(z^{-1}) \tag{1.63}$$

y E_j y F_j se obtienen resolviendo la siguiente ecuación diofántica:

$$1 = E_j(z^{-1})A(z^{-1})\Delta + z^{-j}F_j(z^{-1}) \tag{1.64}$$

El objetivo de la política de control predictiva es hacer que las futuras salidas del sistema $y(t+j)$ sigan a la señal de referencia de valor constante w. Para asegurar una transición suave desde y hasta w se redefine otra señal de referencia mediante la ecuación:

$$w(t+j) = \alpha w(t+j-1) + (1-\alpha)w \tag{1.65}$$

para j=1,2,… siendo α una constante.

El problema de control óptimo consiste en encontrar la política de control que minimice el siguiente índice:

$$J(N,NU)=\sum_{j=1}^{N}\left[y(t+j)-w(t+j)\right]^2+\sum_{j=1}^{NU}\lambda(j)\left[\Delta u(t+j-1)\right]^2 \tag{1.66}$$

donde w es la salida de referencia, N es el horizonte de predicción de salida, NU es el horizonte de predicción de control y λ es el peso del comando.

El resultado de la minimización es

$$\Delta u(t)=\left(\boldsymbol{G}^{\mathrm{T}}\boldsymbol{G}+\lambda\boldsymbol{I}\right)^{-1}\boldsymbol{G}^{\mathrm{T}}(\underline{w}-\underline{f}) \tag{1.67}$$

donde

$$\mathbf{G}=\begin{pmatrix} g_o & 0 & \cdots & 0 \\ g_1 & g_o & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_{N-1} & g_{N-2} & \cdots & g_o \end{pmatrix} \tag{1.68}$$

y los vectores

$$\underline{f}=\begin{pmatrix} f(t+1) & f(t+2) & \cdots & f(t+N) \end{pmatrix}^{\mathrm{T}} \tag{1.69}$$

$$\underline{w}=\begin{pmatrix} w(t+1) & w(t+2) & \cdots & w(t+N) \end{pmatrix}^{\mathrm{T}} \tag{1.70}$$

donde los coeficientes del vector $\underline{f}$ se obtienen como:

$$f(t+1)=\left[G_1(z^{-1})-g_0\right]\Delta u(t)+F_1 y(t) \tag{1.71}$$

$$f(t+2)=z\left[G_2(z^{-1})-z^{-1}g_1-g_0\right]\Delta u(t)+F_2 y(t) \tag{1.72}$$

y así sucesivamente. Como hemos dicho sólo es de utilidad en cada etapa la primera señal de control, entonces

$$u(t)=u(t-1)+\bar{g}^{\mathrm{T}}(\underline{w}-\underline{f}) \tag{1.73}$$

donde $\bar{g}^{\mathrm{T}}$ es la primera fila de $(\mathbf{G}^{\mathrm{T}}\mathbf{G}+\lambda\mathbf{I})^{-1}\mathbf{G}^{\mathrm{T}}$. La habilidad del GPC para producir leyes de control estables para sistemas de fase no mínima es debido a que se asume que los controles aplicados después del horizonte de control NU son constantes, esto es

$$\Delta u(t+j-1)=0 \tag{1.74}$$

Se puede ver que para el caso particular de que $NU=N$, $N\to\infty$ y $\lambda>0$ estaremos ante un RDLC.

Avances más recientes en este tipo de controladores predictivos han ido orientados hacia:

- Mejorar las propiedades de robustez y estabilidad del GPC. [Kouv 92] [Robi 91]
- Formulación de nuevos algoritmos para sistemas MIMO. [Demi 92]
- Mejorar el acercamiento de la salida del sistema a la señal de referencia. [Demi 93]

1.3 La Programación Dinámica

La programación dinámica (PD) [Pun 69] [Bert 87] [Lars 78] [Whit 86] [Snie 92] es una herramienta muy potente de resolución de problemas de optimización tanto discretos como continuos. Está basada en el conocido principio de optimalidad de Bellman [Bell 62], que dice así:

> *"Una política óptima tiene la propiedad de que, sin importar las decisiones previas que se han tomado, el resto de las decisiones deben constituir una política óptima independientemente del estado resultante de las decisiones previas."*

El principio de optimalidad juega un papel similar al que juega el de Pontryagin en el método variacional: sirve para limitar el número de estrategias de control óptimo potenciales que deben ser investigadas. También implica que esta estrategia debe ser determinada trabajando hacia atrás a partir del estado final.

1.3.1 Programación Dinámica para Sistemas Discretos

La programación dinámica da solución al PCOD. Se plantea una función de costo de la forma

$$J = \sum_{k=0}^{N} L(\underline{x}_k, \underline{u}_k, k) \tag{1.75}$$

y unas restricciones generales

$$\underline{x} \in X(k) \subset \mathsf{R}^n \tag{1.76}$$

$$\underline{u} \in U(\underline{x}, k) \subset \mathsf{R}^m \tag{1.77}$$

donde $X(k)$ es el conjunto de los estados admisibles en la etapa k y $U(\underline{x},k)$ el conjunto de controles admisibles en el estado $\underline{x}$ de la etapa k.

La solución se obtiene utilizando una ecuación funcional iterativa que determina el control óptimo para cualquier estado admisible en cualquier etapa. Se define la función de costo mínimo como el mínimo costo que es posible obtener en la trayectoria desde el estado $\underline{x}$ de la etapa k hasta la etapa final N. Su expresión es

$$I(\underline{x},k)=\min_{\underline{u}_k,\underline{u}_{k+1},\cdots,\underline{u}_N}\left\{\sum_{j=k}^{N}L(\underline{x}_j,\underline{u}_j,j)\right\} \tag{1.78}$$

para todo $\underline{x}\in X$ y para todo $k=0,1,\ldots,N$. Aplicando el principio de Bellman se llega a que

$$I(\underline{x},k)=\min_{\underline{u}}\left\{L(\underline{x},\underline{u},k)+I\left[\underline{f}(\underline{x},\underline{u},k),k+1\right]\right\} \tag{1.79}$$

De esta manera se determina $I(\underline{x},k)$, $\forall\underline{x}\in X$ y $\forall k$ tal que $0\le k\le N-1$, a partir del conocimiento de $I(\underline{x},k+1)$ $\forall\underline{x}\in X$. Para completar la solución debemos encontrar $I(\underline{x},N)$. De la ecuación (1.78) vemos que

$$I(\underline{x},N)=\min_{\underline{u}}\left\{L(\underline{x},\underline{u},N)\right\} \tag{1.80}$$

La función de control óptimo será el valor de $\underline{u}$ que conduce al valor mínimo en la obtención de $I(\underline{x},k)$, formalmente

$$\underline{u}^{o}(\underline{x},k)=\arg\min_{\underline{u}}\left\{L(\underline{x},\underline{u},k)+I\left[\underline{f}(\underline{x},\underline{u},k),k+1\right]\right\} \tag{1.81}$$

De esta manera tenemos determinado el control óptimo para cualquier estado en cualquier etapa.

1.3.1.1 El Regulador Discreto Lineal Cuadrático

Para resolver el problema del RDLC mediante la PD [Jacq 81] debemos plantear la función de costo mínimo y minimizar respecto al comando para obtener la política de control óptimo. Las expresiones para una etapa k genérica, así obtenidas, serán las siguientes

$$I(\underline{x},k)=\frac{1}{2}\underline{x}^{\mathrm{T}}\mathbf{P}_k\underline{x} \tag{1.82}$$

$$\underline{u}^{o}(\underline{x},k)=-\mathbf{F}_k\underline{x} \tag{1.83}$$

para $k=N-1,\ldots,0$, donde

$$\mathbf{P}_k=\mathbf{A}_k^T\mathbf{P}_{k+1}\mathbf{A}_k-\mathbf{F}_k^T(\mathbf{R}_k+\mathbf{B}_k^T\mathbf{P}_{k+1}\mathbf{B}_k)\mathbf{F}_k+\mathbf{Q}_k \tag{1.84}$$

$$\mathbf{F}_k=\left[\mathbf{R}_k+\mathbf{B}_k^T\mathbf{P}_{k+1}\mathbf{B}_k\right]^{-1}\mathbf{B}_k^T\mathbf{P}_{k+1}\mathbf{A}_k \tag{1.85}$$

con la condición final $\mathbf{P}_N=\mathbf{S}_N$. Estas ecuaciones se pueden resolver secuencialmente hacia atrás para producir las matrices ganancia de realimentación $\mathbf{F}_{N-1},\mathbf{F}_{N-2},\ldots,\mathbf{F}_0$, así como la secuencia de matrices $\mathbf{P}_{N-1},\mathbf{P}_{N-2},\ldots,\mathbf{P}_0$. Con ello tenemos definida la política de control óptima para cualquier estado en cualquier etapa previamente a la realización del control, es decir, es un proceso *off-line*. Además, por la forma en que se ha hecho la definición,

se puede obtener fácilmente el costo total de la trayectoria óptima desde cualquier punto inicial $\underline{x}_0$ calculando

$$I(\underline{x}_o, k) = \frac{1}{2}\underline{x}_0^{\mathrm{T}}\mathbf{P}_0\underline{x}_0 \tag{1.86}$$

En el caso de que el número de etapas se haga muy grande, la secuencia de matrices de ganancia queda estacionaria, por lo cual se puede utilizar dicho valor estacionario **F** si se desea un diseño de control que no varíe con el tiempo.

Se puede demostrar que la expresión de $\mathbf{F}_k$ es equivalente a la de la matriz $\mathbf{K}_k$ presentada en el apartado 1.2.2, lógico, ya que la solución óptima es única independientemente del método de resolución.

1.3.2 Programación Dinámica hacia Delante

A pesar de que las relaciones recurrentes presentadas en el apartado anterior son apropiadas en la mayoría de problemas, existen ciertos casos en que es deseable usar procedimientos en los cuales la dirección del barrido en la variable de etapa sea al contrario. En estos casos la recursión opera hacia delante en la variable de etapa por lo que estos métodos se denominan usualmente programación dinámica hacia delante.

Existen dos tipos de programación dinámica hacia delante. El primero simplemente renombra la variable de etapa de manera que represente el número de etapas que faltan en vez del índice de la etapa actual. Esta interpretación es completamente aceptada para sistemas invariables pero no se extiende inmediatamente a problemas que varíen con la etapa.

La otra forma de programación dinámica hacia delante es la principal. En este caso definimos la función de costo mínimo $I'(\underline{x},k)$ como el mínimo costo necesario para, comenzando en un estado inicial admisible, llegar al estado $\underline{x}$ en la etapa k. De la definición podremos escribir

$$I'(\underline{x},k) = \min_{\underline{u}_0,\underline{u}_1,\cdots,\underline{u}_{k-1}} \left\{ \sum_{j=0}^{k-1} L(\underline{x}_j, \underline{u}_j, j) \right\} \tag{1.87}$$

Hacer notar que ahora $\underline{x}$ es el último estado de la trayectoria, en vez del estado inicial como en el caso del método hacia atrás.

Para obtener la relación de recurrencia debemos definir la función $\underline{f}^{-1}(\underline{x}_{k+1},\underline{u}_k,k)$ que nos proporciona el estado $\underline{x}_k$ desde el cual se alcanza el estado $\underline{x}_{k+1}$ aplicando el control $\underline{u}_k$. Aplicando el principio de Bellman se llega a que

$$I'(\underline{x},k) = \min_{\underline{u}} \left\{ L\left[\underline{f}^{-1}(\underline{x},\underline{u},k-1),\underline{u},k-1\right] + I'\left[\underline{f}^{-1}(\underline{x},\underline{u},k-1),\underline{u},k-1\right] \right\} \tag{1.88}$$

este procedimiento determina $I'(\underline{x},k)$ en términos de $I'(\underline{x},k-1)$, por lo tanto los cálculos proceden hacia delante en la variable de etapa, en vez de hacia atrás. Para comenzar el proceso debemos determinar la función de costo mínimo en el estado inicial. Si el estado inicial está especificado como $\underline{x}_0$ definimos

$$I'(\underline{x},0)=\begin{cases}0 & si \quad \underline{x}=\underline{x}_0\\ \infty & si \quad \underline{x}\neq\underline{x}_0\end{cases} \tag{1.89}$$

Alternativamente si existe una función de costo inicial dada $\psi(\underline{x})$, entonces hacemos

$$I'(\underline{x},0)=\psi(\underline{x}) \tag{1.90}$$

La correspondiente función de control óptimo $\underline{u}'^{o}(\underline{x},k)$ es la formada por los valores de $\underline{u}$ que minimizan la ecuación (1.88); formalmente

$$\underline{u}'^{o}(\underline{x},k)=\arg\min_{\underline{u}}\left\{L\left[\underline{f}^{-1}(\underline{x},\underline{u},k-1),\underline{u},k-1\right]+I'\left[\underline{f}^{-1}(\underline{x},\underline{u},k-1),\underline{u},k-1\right]\right\} \tag{1.91}$$

función que nos dice que control óptimo debemos aplicar en la etapa $k-1$ para alcanzar el estado $\underline{x}$ en la etapa k.

Una cuestión interesante que surge es la determinación de las condiciones bajo las cuales se obtiene la misma solución que en el caso de la PD normal. En primer lugar, para un problema con el estado inicial y final fijo, ambos métodos obtendrán exactamente la misma secuencia de control óptima y trayectoria óptima. En segundo lugar, para un problema en el cual ni el estado inicial ni el final están especificados pero se dan las mismas funciones de costo iniciales y finales, ambos métodos nuevamente determinarán la misma secuencia de control óptimo y trayectoria óptima, incluyendo el mismo estado inicial y final.

1.3.3 Programación Dinámica para Sistemas Continuos

El principio del óptimo de Bellman puede aplicarse, de igual forma, a sistemas continuos [Lars 82] [Lewi 86]. Sea, entonces, un sistema continuo, modelado por su ecuación diferencial, y un índice de calidad generalizado

$$J=\int_{t_i}^{t_f}L(\underline{x},\underline{u},t)dt+S\left[\underline{x}(t_f),t_f\right] \tag{1.92}$$

Estamos interesados en determinar el control $\underline{u}^{o}(t)$ en el intervalo $[t_i,t_f]$ que minimice J y que conduzca al sistema, desde el estado inicial $\underline{x}(t_i)$, al estado final que satisface

$$\psi(\underline{x}(t_f),t_f)=0 \tag{1.93}$$

Para el estado $\underline{x}$ en el tiempo t se define la función de costo mínimo como

$$I(\underline{x},t)=\min_{\underline{u}(\tau),t\le\tau\le t_f}\left\{S(\underline{x},t_f)+\int_t^{t_f}L(\underline{x},\underline{u},\tau)d\tau\right\} \tag{1.94}$$

Aplicando el principio de Bellman en un intervalo infinitesimal de tiempo llegamos a la igualdad

$$-\frac{\partial I}{\partial t}=\min_{\underline{u}(t)}\left\{L(\underline{x},\underline{u},t)+\left(\frac{\partial I}{\partial \underline{x}}\right)^{\mathrm{T}}\underline{f}\right\} \tag{1.95}$$

Ésta es una ecuación diferencial, para el costo mínimo, denominada de Hamilton-Jacobi-Bellman. Se resuelve hacia atrás en el tiempo desde $t=t_f$ con la condición de contorno

$$I(\underline{x}(t_f),t_f)=S(\underline{x}(t_f),t_f)\quad\text{en el hiper plano}\quad\psi(\underline{x}(t_f),t_f)=0 \tag{1.96}$$

La ecuación (1.95) da solución a el PCOG para sistemas no lineales generales. De cualquier manera, en la mayoría de los casos, es imposible resolverla analíticamente. Cuando puede ser resuelta proporciona la política de control óptimo en lazo cerrado. De ella, formalmente, se pueden deducir los resultados del cálculo variacional y, de forma similar a como ocurrió en el caso discreto, puede resolver el POLC.

La otra posibilidad de aplicación de la PD a sistemas continuos es mediante el método computacional que presentamos a continuación.

1.3.4 Método Computacional

En los apartados anteriores hemos visto el planteamiento teórico de la PD. En la mayoría de los casos las distintas minimizaciones no pueden hacerse analíticamente y debemos proceder a realizar los cálculos numéricamente. Ya que en este trabajo trataremos con sistemas continuos usaremos el método computacional de la PD, por ello nos detendremos algo más en su explicación.

La variable de etapa es la que determina el orden en la que los sucesos ocurren en el sistema. Esta cantidad varía monótonamente durante el periodo en el cual se realizan los controles. En numerosos casos se toma al tiempo como dicha variable. Hasta ahora hemos asumido que la variable de etapa es discreta, para tratar el caso en que la variable de etapa es continua se suele cuantizar en incrementos uniformes. Si denominamos t a la variable de etapa continua, la cual está definida en el rango $t_i\le t\le t_f$, y denominamos Δt al tamaño del incremento, podemos cuantizar las etapas indexándolas mediante la secuencia discreta $k=0,1,\ldots,N$ donde el valor de t correspondiente a k viene dado por

$$t=t_i+k\Delta t \tag{1.97}$$

y donde

$$N\Delta t = t_f - t_i \tag{1.98}$$

Debemos asumir también que *X(k)*, el conjunto de estados admisibles, está cuantizado a un número finito de valores

$$X(k) = \left\{ \underline{x}^1, \underline{x}^2, \ldots, \underline{x}^{P(k)} \right\} \tag{1.99}$$

que suelen elegirse mediante un incremento uniforme. También se asume una cuantización similar para $U(\underline{x},k)$

$$U(\underline{x},k) = \left\{ \underline{x}^1, \underline{x}^2, \ldots, \underline{x}^{M(\underline{x},k)} \right\} \tag{1.100}$$

1.3.4.1 Primer Barrido

Con estas consideraciones, para realizar la minimización de la ecuación (1.79) debemos evaluar la cantidad dentro de las llaves para cada $\underline{u} \in U$ y comparar esos valores directamente para determinar el menor. En este proceso son necesarios dos pasos. Primero se debe evaluar el costo para el estado actual con $L(\underline{x},\underline{u},k)$; en segundo lugar se debe calcular el estado siguiente mediante $\underline{f}(\underline{x},\underline{u},k)$ y con ello evaluar $I(f(\underline{x},\underline{u},k),k+1)$. Vemos que estos cálculos se deben realizar para cada control cuantizado en cada estado cuantizado de cada etapa.

El proceso completo comenzará en la etapa final *N* determinando $I(\underline{x},N) \ \forall \underline{x} \in X(N)$

$$I(\underline{x},N) = \min_{\underline{u} \in U(\underline{x},N)} \left\{ L(\underline{x},\underline{u},N) \right\} \tag{1.101}$$

Si la función de costo no depende del control en la etapa *N*, tendremos directamente que

$$I(\underline{x},N) = L(\underline{x},N) \tag{1.102}$$

Considerando cada estado cuantizado $\underline{x}$ de la etapa *N-1*, se le aplicará cada uno de los controles admisibles $\underline{u}^m \in U(\underline{x},N-1)$. Para cada control se determina el costo de la etapa actual

$$L^m = L(\underline{x},\underline{u}^m,N-1) \tag{1.103}$$

a continuación, para cada uno de esos controles se determina el estado que se alcanza en la etapa *N*

$$\underline{x}^m = \underline{f}(\underline{x},\underline{u}^m,N-1) \tag{1.104}$$

Este estado puede caer fuera del rango de estados admisibles, en este caso el control correspondiente es descartado como candidato a control óptimo para este estado en esta etapa. Si, en cambio, el estado resultante cae dentro del rango permitido pero no en un valor cuantizado, es necesario usar algún procedimiento de interpolación para determinar

$I(\underline{x}^m,N)$ basándonos en los valores de la función de costo mínimo para los estados cuantizados, es decir

$$I(\underline{x}^m,N)=P\left[\underline{x}^m,N,\left\{I(\underline{x},N),\forall\underline{x}\in X(N)\right\}\right] \tag{1.105}$$

El costo total resultante de aplicar el control $\underline{u}^m$ será

$$F^m=L(\underline{x},\underline{u}^m,N-1)+I(\underline{x}^m,N) \tag{1.106}$$

cantidad a minimizar por la elección de $\underline{u}^m$, minimización que se puede realizar simplemente comparando las m cantidades. El valor mínimo, así obtenido, corresponderá a $I(\underline{x},N-1)$ y el control $\underline{u}^m$ para el cual se alcanza el mínimo será el control óptimo $\underline{u}^o(\underline{x},N-1)$.

Este procedimiento se repite para cada estado cuantizado en la etapa $N-1$. Cuando esto está hecho, $I(\underline{x},N-1)$ y $\underline{u}^o(\underline{x},N-1)$ se conocen $\forall\underline{x}\in X(N-1)$. Ahora es posible proseguir con la etapa N-2 y así sucesivamente hasta que se obtengan $I(\underline{x},0)$ y $\underline{u}^o(\underline{x},0)$.

1.3.4.2 Segundo Barrido

El segundo barrido consiste en determinar la secuencia de controles óptimos si partimos del estado $\underline{x}_0$. El primer control de la secuencia será

$$\underline{u}_0^o=\underline{u}^o(\underline{x}_0,0) \tag{1.107}$$

El siguiente estado de la trayectoria se obtiene aplicando la ecuación en diferencias del sistema

$$\underline{x}_1^o=\underline{f}(\underline{x}_0,\underline{u}_0^o,0) \tag{1.108}$$

Si este estado es uno de los cuantizados, el siguiente control óptimo $\underline{u}_1^o$ se obtiene directamente evaluando $\underline{u}^o(\underline{x}_1^o,1)$. En caso contrario $\underline{u}_1^o$ se debe calcular mediante una fórmula de interpolación adecuada basándonos en los controles óptimos conocidos para los estados cuantizados.

Este procedimiento continúa hasta que se obtiene la secuencia de control completa $\underline{u}_0^o,\ldots,\underline{u}_N^o$ y la trayectoria óptima $\underline{x}_0,\underline{x}_1^o,\ldots,\underline{x}_N^o$.

1.3.4.3 Propiedades del Método Computacional

Son varias las propiedades del procedimiento computacional de la programación dinámica. En primer lugar no es necesario asumir ninguna propiedad analítica sobre la ecuación en diferencias del sistema ni de la función de costo de cada estado. Por lo tanto el procedimiento puede acomodarse a sistemas altamente no lineales así como aquellos que se modelan con operaciones lógicas.

Otra propiedad es la facilidad con que se manejan las restricciones. Éstas pueden ser de tipo general, siendo dependientes con la etapa e incluso, los comandos admisibles, dependientes del estado.

Una tercera propiedad es que siempre se determina el mínimo absoluto. Esto es debido a que son considerados todos los estados y todos los controles admisibles. Por ello, dentro de la exactitud de la cuantización, se obtiene siempre el óptimo global.

Otra propiedad favorable es la inherente simplicidad. Los únicos cálculos necesarios son los de la ecuación en diferencias del sistema, la interpolación de la función de costo mínimo y la comparación de magnitudes escalares. Esta simplicidad hace su implementación en computadora bastante sencilla.

Y quizás la propiedad más importante, es que el control obtenido se define en base al estado y la etapa, es decir, implementando un esquema en lazo cerrado. En caso de desviación de la trayectoria óptima el control óptimo real se puede calcular para el resto de los estados.

1.3.4.4 Requerimientos Computacionales

A pesar de las propiedades que presenta la PD su utilización está limitada porque los requerimientos computacionales crecen rápidamente con la dimensión del sistema y el número de variables de control. Este fenómeno fue denominado por Bellman *"la maldición de la dimensionalidad"*. Esto afecta tanto a la complejidad espacial, es decir el número de posiciones de memoria necesarias, así como a la complejidad temporal, el tiempo de cálculo necesario.

Si $P(i)$ es el número de valores cuantizados en la i-ésima variable de estado y siendo n el número total de variables de estado, tendremos una rejilla de

$$P_T = \prod_{i=1}^{n} P(i) \tag{1.109}$$

estados en cada etapa. Suponiendo que $P(i)=P$ para $i=1,\ldots,n$, vemos que el número total de puntos en cada etapa será P^n, es decir, exponencial con la dimensión del sistema. El número de posiciones de memoria necesarias para almacenar los resultados de la primera pasada de la PD será del orden de $N \cdot P^n$.

En cuanto al tiempo de cálculo requerido, vimos que existía una serie de operaciones que era necesario realizar para cada comando en cada estado de cada etapa. Si denominamos Δt_c al tiempo de calculo de esas operaciones, el tiempo total será

$$T_c = M \cdot P_T \cdot N \cdot \Delta t_c \tag{1.110}$$

donde hemos supuesto, por simplicidad, que el numero de comandos discretizados es igual a M en todos los casos. Por lo dicho en el párrafo anterior, tendremos que el

tiempo de calculo es del orden de $M \cdot N \cdot P^n$, exponencial también con la dimensión del sistema.

1.3.5 Implementación de la PD en Tiempo Real

La solución aportada por la PD conduce a una configuración de lazo cerrado. Ésta, junto con otras propiedades, sugieren que la aplicación de la PD a sistemas reales conducirá a buenos resultados.

Un método de implementación es sencillamente almacenar todos los valores de $\underline{u}^o(\underline{x},k)$ en memoria, monitorizar el estado del sistema y buscar el valor apropiado de $\underline{u}^o(\underline{x},k)$. Este tipo de implementación es atractiva porque los cálculos se realizan previamente, la única operación necesaria durante el intervalo de control es localizar el control óptimo. Sin embargo, el principal inconveniente es el gran requerimiento de memoria si deseamos una discretización suficientemente fina o si crece la dimensión del sistema. Con este esquema la complejidad se pasa totalmente al campo espacial.

Un esquema alternativo es realizar la programación dinámica en tiempo real. Al igual que antes, monitorizamos el estado del sistema, pero la secuencia de controles óptimos se recalculan sobre la marcha en una computadora. Normalmente, basándose en las últimas trayectorias calculadas, se reduce el tamaño de los conjuntos X y U sobre los que se hacen los cálculos. En este caso, los requerimientos de memoria no son tantos, pero si es necesario una importante velocidad de cálculo. Se ha pasado la complejidad al campo temporal.

Otro método de utilizar los resultados de la PD es realizar una política subóptima. Una posibilidad es buscar funciones sencillas que aproximen $\underline{u}^o(\underline{x},k)$. La función de costo mínimo del problema original servirá para evaluar la pérdida de rendimiento del controlador subóptimo.

Un aspecto importante de la aplicación en sistemas reales es que la lectura de las variables que componen el estado del sistema se realiza a través de conversores analógicos/digitales (A/D). Estos conversores se encargan de discretizar los posibles valores de las variables según la resolución de que dispongan. Si se ajusta la discretización considerada en el primer barrido de la PD a la que posteriormente se encontrará en la aplicación real se evitará la necesidad de interpolar en el segundo barrido. Esto es así porque en cada etapa el estado que proporcionan los conversores será uno de los considerados y dispondremos para él de su comando óptimo.

2

Redes Neuronales

En este capítulo haremos una presentación de las Redes Neuronales. Por ser un tema relativamente novedoso y aún poco formalizado nos extenderemos para concretar la nomenclatura que utilizaremos. Comenzaremos por una introducción y repaso histórico de éstas en general. A continuación intentaremos dar el esquema general de las redes artificiales nombrando todas sus variantes. En la segunda mitad del capítulo nos centraremos en el tipo de redes utilizadas en este trabajo. Mostraremos su estructura y los dos algoritmos más utilizados para su entrenamiento.

2.1 Introducción

Las redes neuronales se pueden definir como sistemas computacionales, tanto hardware como software, que imitan las habilidades computacionales de los sistemas biológicos usando gran número de neuronas artificiales interconectadas. Las neuronas artificiales son emulaciones sencillas de las neuronas biológicas: toman información desde sensores u otras neuronas artificiales, realizan una operación muy sencilla con

estos datos, y pasan el resultado a otras neuronas artificiales. La organización de estos elementos artificiales está relacionada con la anatomía del cerebro.

Aparte de esta semejanza superficial, las redes neuronales artificiales (RNA) exhiben un sorprendente número de características cerebrales:

- **Aprendizaje.** Las redes neuronales artificiales pueden modificar su comportamiento en respuesta a su entorno. Este factor, más que ningún otro, es el responsable del interés que han recibido. Mostrado un conjunto de entradas, quizás con sus correspondientes salidas deseadas, ellas se autoajustan para producir salidas consistentes. Se han desarrollado una amplia variedad de algoritmos de entrenamiento, cada uno de los cuales con sus propias virtudes y defectos.
- **Generalización.** Una vez entrenada, la respuesta de la red puede ser insensible, en cierto grado, a las variaciones de la entrada. Esta habilidad de ver, a pesar del ruido y las distorsiones, los patrones que subyacen, es vital para el reconocimiento de patrones en un entorno real. Las redes neuronales producen un sistema que puede tratar con el imperfecto mundo real en que vivimos superando la ceguera de las computadoras convencionales. Es importante hacer notar que las redes neuronales artificiales generalizan automáticamente como resultado de su estructura, no por usar la inteligencia humana que se deriva de la de un programa hecho *ad hoc*.
- **Abstracción.** Algunas redes neuronales son capaces de abstraer la esencia de un conjunto de entradas. Por ejemplo, una red puede ser entrenada en una secuencia de versiones distorsionadas de la letra A. Tras un entrenamiento adecuado, la aplicación de ese ejemplo distorsionado hará que la red produzca una letra perfectamente formada. En esencia, se ha entrenado para producir algo que nunca ha visto antes.
- **Tolerancia a fallos.** Las redes neuronales son el primer método computacional disponible inherentemente tolerante a fallos. La razón de esto es la manera distribuida y redundante en que se codifica la información. La mayoría de los algoritmos y métodos de almacenamiento de datos depositan cada pieza de información en una posición única, localizada y direccionable. Cuando las redes neuronales almacenan información, ésta suele estar no localizada. En cambio, las numerosas interconexiones entre los nodos de la red tomarán valores tales que, cuando la red es estimulada convenientemente, generará un patrón de salida que representa la información almacenada. Dado un estímulo diferente, la misma red con los mismos pesos, producirá una salida diferente. Por ello, a

pesar de que parte de la red sea destruida, pueden seguir actuando sólo con degradaciones progresivas.

- **Operación en Tiempo real.** En numerosas aplicaciones la principal prioridad es la necesidad de procesar grandes cantidades de datos rápidamente. Las redes neuronales son adecuadas para la implementación paralela. Por su estructura, se deben realizar unos pocos pasos en cada neurona. Para la mayoría de las operaciones de las redes, la necesidad de entrenamiento, único aspecto que consume tiempo, es mínima en un entorno de tiempo real.
- **Facilidad de inserción en tecnología existente.** Una red individual puede ser entrenada para realizar una única tarea bien definida. Ya que una red puede ser rápidamente diseñada, entrenada, probada, verificada, y trasladada a una implementación hardware de bajo costo, es fácil insertar redes neuronales de propósito específico dentro de sistemas existentes. De esta manera, las redes neuronales pueden ser usadas para mejoras y actualizaciones incrementales de sistemas, y cada paso puede ser evaluado antes de continuar con sucesivos desarrollos.

Las redes neuronales ofrecen aumento del rendimiento con respecto a las técnicas convencionales [Mill 90] [Nare 90] [Hunt 91] [Tayl 92] [Cich 93] [More 93a] [More 95] en áreas que incluyen, entre otras:

- Reconocimiento de patrones.
- Filtrado de señales.
- Clasificación de datos.
- Compresión de datos.
- Búsqueda asociativa.
- Control adaptativo.
- Optimización y control óptimo.

2.2 Historia de las Redes Neuronales

Las redes neuronales parecen ser un desarrollo reciente, sin embargo, este campo fue establecido antes de la llegada de las computadoras, y ha atravesado varias épocas [Mare 90] [Neur 91].

2.2.1 Primeros Fundamentos

En el siglo XVIII William James indicó que en el aprendizaje eran importantes las múltiples conexiones. Definió la reintegración como el proceso en el cual la información perdida se reconstruye y produce un recuerdo total, concepto estrechamente relacionado

con las memorias asociativas. También indicó que la actividad de la neurona estaba relacionada con la suma de sus entradas. Este trabajo no inspiró ninguna simulación pero estableció la opinión de su tiempo con respecto al cerebro.

En 1943 McCulloch y Pitts desarrollaron modelos de redes neuronales basándose en sus conocimientos de neurología [Cowa 93] [Greg 95]. Estos modelos hacían varias suposiciones de cómo funcionaban las neuronas. Sus redes, de estructura fija, estaban basadas en neuronas simples, las cuales eran consideradas dispositivos binarios con umbrales fijos. Las sinapsis de conexión tenían pesos idénticos y las sinapsis inhibitorias bloqueaban la transmisión completamente. Sus modelos también incluían el efecto de retardo de la sinapsis. Este modelo, aunque es capaz de implementar cualquier expresión lógica finita, se ha reconocido como limitado. Su importancia viene de las múltiples conexiones y del poder potencial de procesadores simples interactuando.

2.2.2 Primeras Simulaciones

Dos grupos publicaron las primeras simulaciones en computadoras de modelos neuronales inspirados por la neurociencia de sus días: Farley y Clark; y Rochester, Holland, Haibt y Duda. Usaban las computadoras para determinar la validez de sus ideas. El grupo de Farley y Clark, de los laboratorios IBM, mantenían un estrecho contacto con Donald Hebb y Peter Milner, neurofisiólogos de la universidad de McGill. Cuando las primeras simulaciones no funcionaban, ellos consultaban a los neurofisiólogos y revisaban su modelo para incluir la última información de los laboratorios neuronales. Tal interacción con neurofisiólogos estableció un tratamiento multidisciplinar el cual continúa hasta el día de hoy. Varios conceptos del laboratorio de Hebb se demostraron valiosos. Se estableció que: la información está almacenada en la fuerza de las interconexiones; la magnitud de los pesos de conexión debe restringirse; la aplicación de inhibición, consistente en reducir los pesos de conexión en condiciones apropiadas; y la sustitución de la naturaleza binaria de las neuronas por una salida de frecuencia de disparo en rango continuo.

Rosenblatt, en 1958, aumentó el interés y actividad en este campo cuando diseñó y desarrolló el perceptrón. El perceptrón está compuesto de tres capas, con la capa intermedia conocida como capa asociativa. Existe un proceso competitivo entre las unidades de la capa de salida, resultando de las conexiones inhibitorias entre dichas unidades, dando lugar a una clasificación exclusiva de los patrones. La contribución de Rosenblatt fue considerada un gran avance. Estableció la naturaleza de las relaciones entre la entrada y la salida con el sistema, consiguiendo con ello clasificaciones distintas y separadas. El perceptrón es computacionalmente preciso y es realmente una máquina que

aprende. A pesar de que el sistema exhibía un comportamiento adaptativo complejo, Rosenblatt reconoció que tenía limitaciones.

La red ADALINE (elemento adaptativo lineal) fue desarrollada, poco después del perceptrón, por Widrow y Hoff en la Universidad de Stanford. Es un dispositivo electrónico analógico compuesto por elementos simples. Tanto el ADALINE como el MADALINE (múltiples ADALINEs) emplean un procedimiento de entrenamiento más sofisticado que el del perceptrón. Esta técnica es llamada LMS (*least mean squares*), también conocida como *Regla Delta* porque trabaja minimizando una diferencia entre la salida observada y la deseada. La reducción de esa delta implica la utilización del algoritmo del gradiente descendente, pero Widrow y Hoff demostraron que no es necesario obtener las derivadas. La mejora importante con respecto al perceptrón es que los pesos son corregidos incluso cuando la salida es correcta, produciendo un entrenamiento más rápido y exacto.

2.2.3 Época de Recesión

A pesar de la aparición del ADALINE, los desarrollos se centraban en el perceptrón. Éste, con una única capa de unidades asociativas, sólo puede aprender a discriminar entre patrones linealmente separables. Cuando esos sistemas se probaban con patrones complejos se observaba poco progreso. En 1969 Minsky y Papert publicaron un libro, llamado *Perceptrons* [Mins 69], en el cual probaban que una red de una sola capa era teóricamente inaplicable para resolver algunos problemas simples (como por ejemplo el de la función O-exclusiva). El análisis que hicieron los autores del perceptrón sencillo fue correcto pero se centraron sobre todo en lo que éste no podía hacer, sin comentar lo que sí podía. El error fundamental de Minsky y Papert fue generalizar las limitaciones de un perceptrón de una única capa a sistemas multicapas. Su frase *"… nuestro juicio intuitivo es que la extensión (a sistemas multicapas) es estéril"* posteriormente se ha demostrado totalmente inadecuada. El resultado significativo de este libro fue la eliminación de los fondos para los desarrollos en simulación de redes neuronales. Además abrió una brecha entre el grupo tradicional de la inteligencia artificial que trabajaba con sistemas expertos y aquellos que trabajaban con redes neuronales. Esto ha sido particularmente perjudicial porque, como se está viendo en numerosas áreas, para obtener mejores resultados en inteligencia artificial está siendo necesario la combinación de ambas aproximaciones.

2.2.4 Innovaciones

A pesar de que el interés público y los fondos disponibles eran mínimos, cierto número de científicos continuaban trabajando para desarrollar métodos computacionales basados en la neurofisiología para resolver problemas como el reconocimiento de

patrones. Estos trabajos proporcionaron nuevos métodos, base de la posterior popularidad y expansión de este campo.

2.2.4.1 Redes Autoorganizativas

En 1967 Shun-Ichi Amari publicó un trabajo en el cual establecía la base matemática para una teoría de aprendizaje que abordaba la clasificación adaptativa de patrones. En 1972 extendió su trabajo mostrando como una red autoadaptativa podía formar un patrón representativo a partir de patrones de estímulo. Él consideró esta red autoadaptativa como un modelo para la memoria asociativa.

2.2.4.2 Memorias Asociativas.

Los trabajos iniciales sobre memoria asociativa fueron publicados en 1972 de manera independiente por James Anderson y Teuvo Kohonen, sin que uno tuviera conocimiento de la contribución del otro.

James Anderson era neurofisiólogo en la Universidad Brown de los Estados Unidos. En 1970 comenzó a desarrollar un modelo lineal, denominado asociador lineal, basado en modelos de almacenaje en la memoria, recuerdo y reconocimento. Posteriormente lo extendió a una red llamada *brain-states-in-a-box*. Sus trabajos son interesantes por su orientación neurofisiológica.

Teuvo Kohonen, de la Universidad Técnica de Helsinki, tenía una formación de ingeniería eléctrica. A principio de los años 70 trabajó en el aprendizaje adaptativo y memorias asociativas produciendo la denominada memoria matricial de autocorrelación. Posteriormente formuló el principio de entrenamiento competitivo y desarrolló las redes denominadas AVQ (*Adaptative Vector Quantization*) y LVQ (*Learning Vector Quantization*). Estas son redes autoorganizativas y crean su propia representación de las categorías de los datos de entrada, es decir, no necesitan un profesor. Están formadas por una capa de neuronas altamente interconectadas, las cuales tienen una relación cooperativa. Kohonen ha desarrollado sus conceptos básicos para crear la TMP (*Topology Preserving Map*) en la cual las neuronas tienen conexiones positivas con sus vecinas más cercanas y negativas con las neuronas más alejadas.

2.2.4.3 Backpropagation

La primera publicación en la técnica de la Backpropagation (BP) fue debida a Paul Werbos el cual comenzó a estudiar, con un fuerte interés, como la mente humana procesaba la información. El método de la BP era un pequeño, aunque esencial, elemento del gran modelo que él propuso en 1974 y al cual describió como una *"forma eficiente*

de calcular derivadas". Para probar que las derivadas calculadas por este método eran exactas y correctas él tuvo que probar un teorema generalizado que denominó la regla de la cadena para derivadas ordenadas.

2.2.4.4 Redes Neuronales Resonantes

En 1988 Carpenter y Grossberg trabajaron juntos en el desarrollo de las redes ART (*Adaptative Resonance Theory*). Su principal interés era demostrar la validez fisiológica de las redes, más que en resolver problemas prácticos. Su primer concepto es que, si una neurona de una población está fuertemente excitada, las que la rodean recibirán una señal inhibitoria. Las redes ART son radicalmente diferentes de las que existían hasta entonces.

Grossberg también contribuyó enormemente a la teoría de las memorias de redes, es decir, como los patrones pueden quedar activos después de que han terminado las entradas a la red. El definió las memorias de término corto (STM) y las memorias de término largo (LTM). En ambos casos los valores de activación y los pesos decrecen con el tiempo, una característica denominada olvido.

Otros elementos originales de Grossberg fueron la utilización de la sigmoide como función de salida de las neuronas y un algoritmo de entrenamiento que se basa en la suma de las salidas actuales y las deseadas, en vez de en la diferencia como hacían Widrow y Hoff.

2.2.4.5 Redes Cooperativas/Competitivas

Por otro lado Kunihiko Fukushima del laboratorio NHK de Japón desarrolló un sistema de red neuronal multicapa capaz de interpretar caracteres escritos a mano, por lo que se trataba de un modelo del sistema neuronal visual. La red original fue publicada en 1975 con el nombre de *Cognitron*. Su grupo también ha publicado, a lo largo de los 80, sobre su sistema mejorado, denominado *Neocognitron*.

La red original era autoorganizativa y aprendía sin supervisión. Las versiones posteriores usaban un entrenamiento supervisado, pero reconocía que esto es más un planteamiento en aplicaciones prácticas que un modelado biológico puro. Como se emulaba el sistema neuronal visual, a partir de la retina, cada capa es bidimensional compuesta por celdas de salida no negativa y continua. La capacidad más interesante de estas redes es que la salida no es afectada por la posición del patrón en el campo de entrada ni por cambios en el tamaño o forma de este patrón.

2.2.5 Reaparición

Los progresos durante los setenta y principios de los ochenta fueron importantes para la reaparición del interés en el campo de las redes neuronales. Varios factores influyeron en este movimiento, entre ellos los libros generales y las conferencias que proporcionaron un forum para personas en diversos campos. En 1979 Hinton, Anderson y Norman organizaron una pequeña conferencia a la que acudieron científicos de diferentes campos: neurofisiólogos, psicólogos, matemáticos, expertos en inteligencia artificial e ingenieros. Como resultado de esta conferencia se publicaron una serie de trabajos denominados *Parallel Models of Associative Memory*.

Entre las publicaciones individuales destaca la de Hopfield en 1982 [Hopf 82]. Mientras que la mayoría del trabajo realizado en los años anteriores fue realizado por científicos relacionados con los seres vivos, Hopfield era un respetado físico. Identificó estructuras de redes y algoritmos que podían ser generalizados y presentaban un alto grado de robustez. Presentó sus redes en una manera que era fácil de entender para ingenieros e informáticos, mostrando las similitudes entre su trabajo y el de otros, y llamando la atención del mundo tecnológico. No introdujo muchas nuevas ideas pero las reunió de una forma diferente.

La red que presentó tenía de original la definición de energía de la red y la forma de reducir esa energía con el entrenamiento. La estabilidad de la dinámica de la red fue descrita matemáticamente por funciones de Lyapunov. Mientras que originalmente sus neuronas eran binarias, en un artículo posterior [Hopf 1984] presentó la versión con salida sigmoide. Estas ideas fueron recogidas por la industria de los semiconductores, en concreto por los laboratorios de la AT&T Bell, que en menos de tres años anunció la primera red neuronal en chip de silicio, la cual utilizaba los algoritmos de Hopfield. La red de Hopfield es actualmente la más conocida de las redes autoasociativas.

En 1986 Rumelhart, McClelland y el Grupo PDP publicaron los primeros dos volúmenes de su *Parallel Distributed Processing* [Rume 86] [McCl 86], a los cuales siguió un tercero en 1988. Los libros presentaban todo lo práctico que había que saber en 1986 de las redes neuronales de una manera entendible, usable e interesante. En ellos se presentaban varios ejemplos de arquitecturas, funciones de transferencia y algoritmos de entrenamiento. Entre sus capítulos cabe destacar el octavo *"Learning Internal Representations by Error Propagation"*, en el que aparece la derivación del algoritmo de la Backpropagation para el perceptrón multicapa; y el séptimo que trata de la maquina de Boltzman. Claramente una de las mayores contribuciones de los volúmenes del PDP ha sido la popularización de la Backpropagation como algoritmo de entrenamiento.

2.3 Tipos de Redes Neuronales

Las habilidades de las redes neuronales artificiales están relacionadas con los elementos, estructura y método de entrenamiento de la red [Free 93]. Las posibilidades en estos parámetros dan lugar a numerosos tipos de redes neuronales, cada una de las cuales tiene diferentes características particulares.

2.3.1 La Neurona Artificial

La operación de una típica neurona artificial se ilustra en la figura 2.1. Esta neurona acepta una señal de entrada o un conjunto de señales procedentes de otras neuronas o dispositivos de entrada. En un único ciclo de operación, suma juntas las señales recibidas, añade o sustrae un valor umbral y pasa este valor sumado a través de una función de activación o función de transferencia. Ésta produce un valor que se denomina activación de la neurona, funcionalmente dependiente del valor suma anterior. Si la función de transferencia produce un valor positivo, o por encima de un nivel predeterminado, la salida es conducida (multiplicada) a través de los llamados pesos hacia otras neuronas.

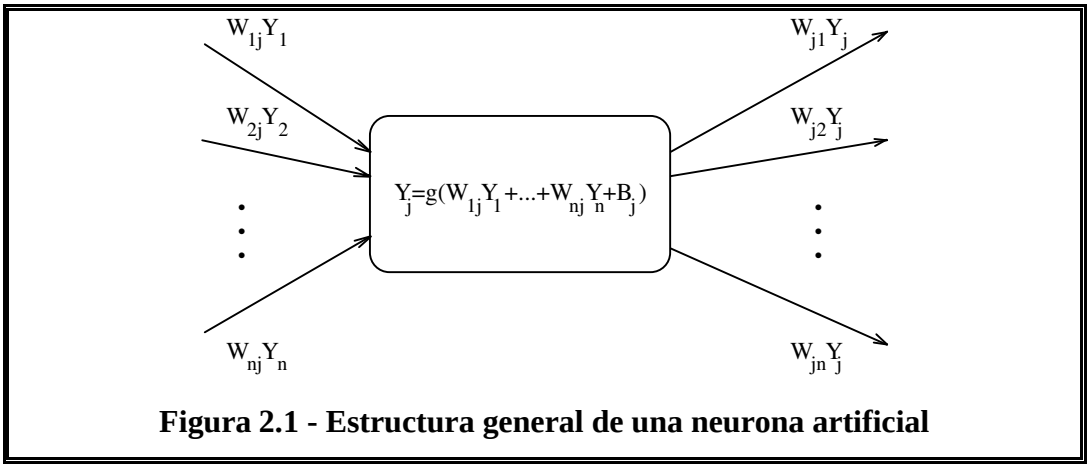

Figura 2.1 - Estructura general de una neurona artificial

La forma más significativa en la cual una neurona artificial se diferencia de las demás incluye: el cambio de la función de transferencia, añadir nuevos parámetros o funciones; y considerar el potencial almacenamiento de información más allá de un único ciclo de operación.

2.3.1.1 Función de Transferencia

Esta función especifica como la neurona escalará su respuesta a señales de entrada y produce la activación de la neurona. Si la activación es suficientemente fuerte la neurona artificial enviará una señal a las neuronas a las que está conectada. Las cuatro funciones de transferencia más usadas son:

- **Función umbral lógica.** La forma más sencilla de definir la activación de una neurona es considerarla binaria, es decir 0 ó 1. Si la suma de las entradas es mayor o igual que el umbral de la neurona, la activación es 1; si es menor, la activación es cero. Las neuronas con este tipo de función son fácilmente implementables en hardware pero tienen capacidades muy limitadas.

- **Función saturación.** En este caso se definen un límite superior e inferior. Si la suma de las entradas es menor que el límite inferior, la activación se define como 0 (ó -1). Si la suma de las entradas es mayor o igual que el límite superior, la activación es 1. Si está entre el límite superior e inferior, la activación se define como una función lineal de la suma de las entradas. Esta función queda definida entonces mediante la ecuación

$$g(z) = \begin{cases} 0 & \text{para} \quad z \le -\alpha \\ z/2\alpha + 1/2 & \text{para} \quad -\alpha < z < \alpha \\ 1 & \text{para} \quad z \ge \alpha \end{cases} \tag{2.1}$$

  donde α es el límite superior y -α es el límite inferior.

- **Función continua.** Las funciones anteriores no son diferenciables, por lo cual, presentan problemas para algoritmos de entrenamiento como el de la BP. Es necesario, entonces, una función continua definida desde menos a más infinito, monótonamente creciente y que tenga límite inferior y superior. La función sigmoide ha sido usada ampliamente, su ecuación es

$$g(z) = \frac{1}{1 + e^{-\alpha z}} \tag{2.2}$$

  donde *z* es la suma de todas las entradas y α es una constante. La forma de la sigmoide produce que, para la mayoría de los valores de las entradas, la salida esté cerca de uno de los valores asintóticos. Esto permite que las salidas estén claramente agrupadas en una de las dos clases: altas o bajas.

  Un factor importante con respecto a la función sigmoide, y otras de forma similar como la tangente hiperbólica, es que su derivada es siempre positiva y próxima a cero para valores muy positivos o muy negativos de *z*; en cambio, alcanza su máximo valor cuando *z* es igual a cero. Esta característica resuelve el dilema ruido-saturación. Para que una red pueda tratar pequeñas y grandes señales: las pequeñas entradas requieren una alta ganancia a través de la red, para ser significativas; mientras que las grandes deben tener poca ganancia para evitar la saturación cuando las neuronas se colocan en cascada.

- **Funciones radiales.** Es un tipo poco común de función de transferencia. De entre ellas la más usada es la función Gaussiana, que es útil cuando la red neuronal se usa para reproducir funciones continuas. La media y la desviación de esta función se puede variar, lo cual la hace más adaptable que la función sigmoide.

2.3.1.2 Sesgo y Ganancias

Algunas arquitecturas neuronales añaden a la suma de las entradas un valor de sesgo (*bias*) que modeliza el umbral de disparo de las neuronas reales. Éste se implementa como un nodo en la capa anterior con salida constante, que usualmente tiene un valor ajustable, que es conectado a nuestro nodo. Esto se utiliza para escalar la suma a un rango útil.

La suma de las entradas puede ser multiplicada por un factor de ganancia. Este factor puede aparecer en la propia función de transferencia. Usualmente posee un valor fijo, aunque en ciertos modelos es adaptable. De igual forma una ganancia adaptable se puede aplicar a la salida de la neurona, esto permite que las salidas de ciertas neuronas sean más fuertes que otras.

2.3.1.3 Memoria

La mayoría de las redes neuronales recuerdan las entradas que han recibido únicamente durante el ciclo de operación actual. Esto es ineficiente para ciertas aplicaciones, sobre todo aquellas en las que se encuentran secuencias temporales en los patrones de entrada o en las que controlan algún proceso que varía con el tiempo. Por lo tanto es útil desarrollar redes neuronales que tengan alguna forma de memoria. Existen dos formas sencillas de hacer esto. La primera es proporcionar una función de memoria dentro de cada nodo, pudiéndose hacer una suma sobre un intervalo de tiempo o utilizando los estados previos en una ecuación dinámica. El segundo método se puede implementar mediante conexiones recurrentes, es decir, alguna de las salidas en el ciclo actual formarán parte de las entradas del ciclo siguiente.

2.3.2 Estructura de la Red

La forma en que se interconectan las neuronas nos permite definir diferentes tipos de arquitecturas. Debemos hacer algunas distinciones básicas para formar las categorías de redes. La primera distinción es el número de capas en la red. Clasificaremos las redes como monocapas, bicapas o multicapas. Dentro del primer grupo, el tipo de conexión permitida crea la posibilidad de diferentes estructuras. Los dos tipos principales de redes monocapas son: aquellas que están lateralmente conectadas y aquellas que tienen sólo un orden lógico. Las redes bicapas típicamente tienen conexiones tanto hacia delante como

hacia atrás. Usualmente no tienen conexiones laterales. Existe una gran clase de redes multicapas que poseen únicamente conexiones hacia delante, y hay un número de redes multicapas con conexiones complejas (hacia delante, hacia detrás y laterales). Basándonos en estas distinciones podemos identificar cinco estructuras diferentes:

- **Redes neuronales multicapas de alimentación hacia delante.** En estas redes las señales se propagan hacia delante a través de las capas. No existen autoconexiones, conexiones laterales o conexiones hacia detrás. Ejemplos de estas redes son el perceptrón y las redes BP. Son las redes más utilizadas y suelen ser elegidas para aplicaciones de clasificación y reconocimiento de patrones.
- **Redes monocapas lateralmente conectadas.** Ya que esta red sólo tiene una capa, sólo puede activar un patrón cada vez. Las conexiones laterales, y a veces recurrentes, causan que en cada ciclo de operación aparezcan diferentes patrones. En cuanto a utilización, están en segundo lugar con respecto a las anteriores. Su uso típico es para patrones de asociación, es decir, que almacenan numerosos patrones pero sólo manifestan uno cada vez. Las redes de Hopfield y Anderson son ejemplos de este tipo.
- **Redes monocapas topográficamente ordenadas.** Estas redes no tienen conexiones explícitas. Durante el entrenamiento, una medida de la distancia vectorial entre diferentes vectores de neuronas se usa para ajustar su posición relativa en el espacio vectorial. Son redes que están creciendo en importancia y en esta clase encontramos las LVQ y las TPM desarrolladas por Kohonen.
- **Redes asociativas/resonantes.** Son redes bicapas en las cuales la información pasa tanto hacia delante como hacia detrás. Para ello tienen conexiones hacia delante y hacia detrás con conjuntos de pesos típicamente diferentes, y no sólo la trasposición de unos con respecto a otros. Este tipo de redes es particularmente buena para la asociación de patrones en la primera capa con otros en la segunda capa, lo que se conoce como heteroasociación de patrones. También pueden ser usadas para clasificación de patrones. Las redes más recientes llevan una dinámica que se denomina resonancia, en la cual los patrones en la primera y segunda capa se estimulan mútuamente hasta que se llega a un estado estable. Las dos más populares son la ART de Carpenter y Grossberg y la BAM de Kosko.
- **Redes multicapas cooperativas/competitivas.** Algunas redes, dentro de la categoría de alimentación hacia delante y alimentación hacia detrás, tienen conexiones laterales. Estas conexiones están diseñadas específicamente de manera que las positivas (cooperativas) compensen a las negativas

(competitivas). Este diseño se ha hecho para imitar ciertas redes biológicas, como por ejemplo el *Boundary Contour System* de Grossberg que es una emulación del proceso de visión biológico.

2.3.3 Tipo de entrenamiento

Una red es entrenada de manera que la aplicación de una serie de entradas produce el conjunto de salidas deseadas, o al menos consistente. Cada una de esas entradas, o salidas, se denomina vector. El entrenamiento se lleva a cabo aplicando secuencialmente vectores de entrada mientras se ajustan los pesos de la red de acuerdo a un procedimiento predeterminado. Durante el entrenamiento, los pesos de la red convergen gradualmente a valores tales que cada vector de entrada produzca el vector de salida deseado. Estos algoritmos se clasifican en supervisados y no supervisados.

2.3.3.1 Entrenamiento Supervisado

En los entrenamientos supervisados un maestro ha de emparejar cada vector de entrada con un vector de salida deseado formando los denominados pares de entrenamiento [Hush 93]. Una red, usualmente, es entrenada sobre un cierto número de esos pares. El proceso consiste en: aplicar un vector de entrada; calcular el vector de salida y compararlo con el vector objetivo correspondiente; alimentar la diferencia (el error) hacia atrás a través de la red; y modificar los pesos según un algoritmo que tiende a minimizar el error. Los vectores del conjunto de entrenamiento son aplicados secuencialmente y se calcula el error y el ajuste de los pesos para cada vector. Esto se repite hasta que el error, para todo el conjunto de entrenamiento, sea aceptablemente bajo.

2.3.3.2 Entrenamiento no Supervisado

A pesar de las múltiples aplicaciones con éxito, el entrenamiento supervisado es criticado por poco factible como modelo biológico. El entrenamiento no supervisado es un modelo mucho más plausible de los procesos de entrenamiento de los sistemas biológicos. Los algoritmos de este tipo no requieren vector objetivo para las salidas, y por lo tanto, no existe comparación con una respuesta ideal predeterminada. El conjunto de entrenamiento consiste únicamente en vectores de entrada. El algoritmo de entrenamiento modifica los pesos de la red para producir salidas que sean consistentes, es decir, dos aplicaciones del mismo vector de entrenamiento, o de un vector que sea suficientemente similar, debe producir el mismo vector de salida. El proceso de entrenamiento extrae las propiedades estadísticas del conjunto de entrenamiento y agrupa vectores similares dentro de clases. Aplicando a la entrada un vector de una clase dada producirá un vector de salida específico, pero no hay manera de determinar, antes del

entrenamiento, que patrón de salida específico producirá una clase de vectores dada. Por ello, la salida de tales redes generalmente debe ser transformada a una forma comprensible tras el periodo de entrenamiento. Esto no es un serio problema y usualmente es una manera simple de identificar la relación establecida por la red.

La mayoría de los algoritmos de entrenamiento están basados en los conceptos de Hebb. El propuso un modelo para el entrenamiento no supervisado, en el cual los pesos de conexión se incrementaban si estaban excitadas tanto la neurona fuente como la destino. De esta manera eran reforzados los caminos frecuentemente utilizados, y se explicaban los fenómenos de hábito y aprendizaje por repetición. Una red neuronal, que use este algoritmo, incrementará sus pesos de acuerdo al producto del nivel de excitación de la neurona fuente y destino.

2.4 Arquitectura de las redes BP

Durante numerosos años no había algoritmo para entrenar redes artificiales multicapas. Desde que se probó que la red de una sola capa estaba severamente limitada todo el campo de las redes neuronales entró en eclipse. La invención del algoritmo de la BP ha jugado un gran papel en la reaparición del interés en las redes neuronales. La BP es un método sistemático, matemáticamente fundamentado, para entrenar redes neuronales multicapas. La BP ha aumentado enormemente el rango de problemas a los cuales se pueden aplicar las redes neuronales, y ha generado numerosas demostraciones con éxito de su poder.

La figura 2.2 muestra la neurona usada como unidad de construcción fundamental de las redes para la BP. Posee un conjunto de entradas que provienen, bien desde la entrada de la red, o bien desde una capa previa. Suele existir el sesgo, que consiste en una entrada conectada a un valor constante de 1. Cada una de las entradas es multiplicada por un peso y los productos son sumados. Esta suma de productos se denomina *z* y debe ser calculada para cada neurona en la red. Una vez calculada *z*, se le aplica una función de activación *g* para producir la señal *y*. La única condición para *g* es que sea una

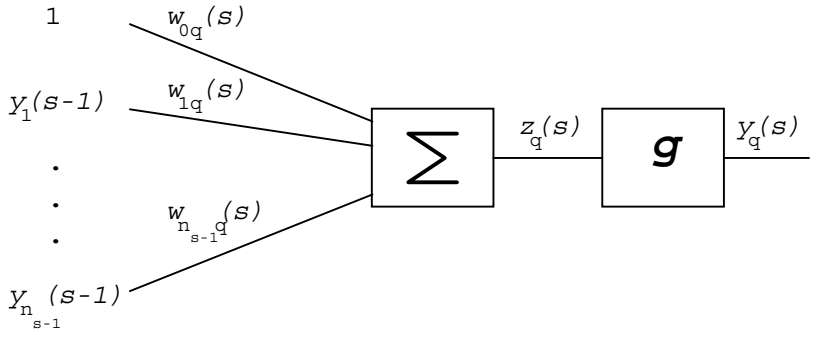

Figura 2.2 - Estructura de la neurona *q* de la capa *s* en una red BP.

función diferenciable y es conveniente que la expresión de su derivada sea simple. Si además esta función se elige no lineal la red multicapa posee un gran poder de representación, en comparación con una red monocapa. Suele utilizarse la función sigmoide (2.2) cuya derivada queda

$$\frac{dg(z)}{dz} = g(z)(1-g(z)) \tag{2.3}$$

Existen numerosas funciones que pueden ser usadas, entre ellas la tangente hiperbólica, cuya derivada queda

$$\frac{dg(z)}{dz} = 1-g(z)^2 \tag{2.4}$$

La estructura de la red adecuada para el entrenamiento con la BP, que suele denominarse perceptrón multicapa, se muestra en la figura 2.3. El primer conjunto de neuronas, conectadas a la entrada, sirven sólo como puntos de distribución. No realizan suma de las entradas sino que simplemente pasan la señal de entrada hacia los primeros pesos. Cada neurona en las capas sucesivas produce las señales *z* e *y*. Algunos autores incluyen la capa de entrada (no actuante) al indicar el número de capas de la red. Otros, en cambio, sólo cuentan las capas de pesos. Nosotros consideraremos la capa de entrada como capa cero. La BP se puede aplicar a redes con cualquier número de capas. Todas las conexiones son hacia delante, de izquierda a derecha en la figura, formando una red de alimentación hacia delante. Es posible aplicar la BP a redes con conexiones hacia detrás pero es muy poco usual. Cada neurona tiene tantas entradas como neuronas la capa anterior y su salida alimenta a todas las neuronas de la capa siguiente, formando una estructura totalmente interconectada.

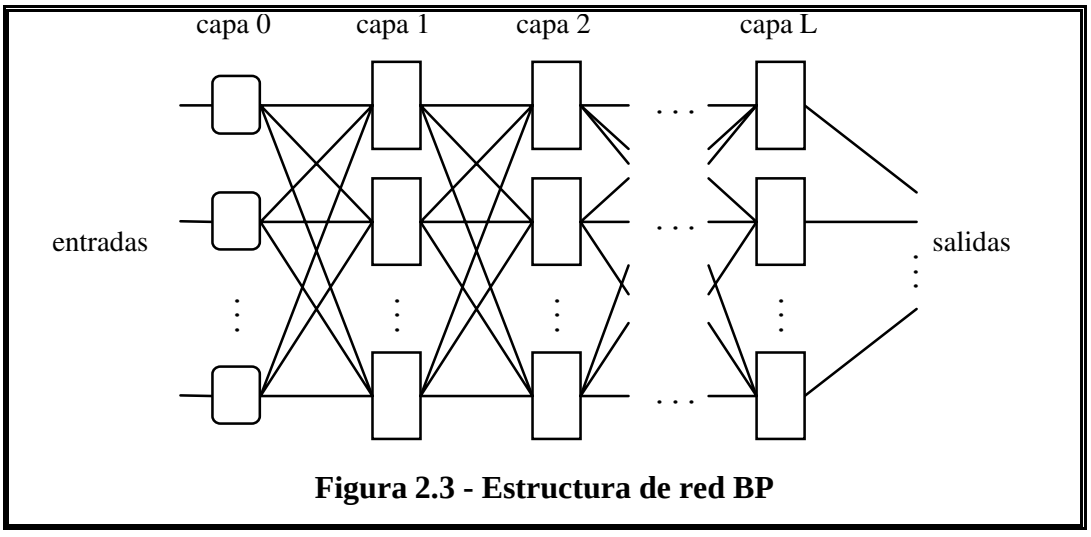

Figura 2.3 - Estructura de red BP

A la vista de la forma de la neurona y la estructura de la red, la ecuación completa de una neurona será

$$z_q(s) = \sum_{p=1}^{n_{s-1}} w_{pq}(s) y_p(s-1) + w_{0q}(s); \quad y_q(s) = g(z_q(s)) \tag{2.5}$$

donde $s=1,...,L$ es el número de la capa y $q=1,...,n_s$ es el cardinal de la neurona dentro de la capa s.

2.5 Algoritmo de la BP

2.5.1 Primera Aproximación al Entrenamiento

El objetivo del entrenamiento de la red es ajustar los pesos de manera que la aplicación de un conjunto de entrada produzca el conjunto de salida deseado. Antes de comenzar este proceso, todos los pesos deben inicializarse a valores aleatorios pequeños. Esto asegura que la red no está saturada por grandes valores en los pesos y previene algunas otras patologías. Por ejemplo, si los pesos empiezan todos con igual valor y las prestaciones necesarias requieren valores diferentes, la red no aprenderá. El proceso de entrenamiento de una red BP será:

Repetir
- Seleccionar el siguiente par de entrenamiento del conjunto
- Aplicar el vector de entrada a las entradas de la red
- Calcular la salida de la red
- Calcular el error entre la salida de la red y la salida deseada
- Ajustar los pesos de la red para minimizar el error

Hasta que el error sea aceptablemente bajo

Las operaciones requeridas en los tres primeros pasos anteriores son similares a las que aparecen al utilizar una red entrenada, esto es, se aplica un vector de entrada y se calcula el vector de salida resultante. Los cálculos se realizan capa por capa. Cada salida de la red es restada de la correspondiente componente del vector objetivo para calcular el error. Finalmente este error es utilizado para ajustar los pesos de la red.

Después de un número suficiente de repeticiones de estos cuatro pasos, que suele ser del orden entre 100 y 10.000 veces el número de pesos de interconexión, el error entre la salida actual y la salida deseada debe haberse reducido a un valor aceptable y se dice que la red está entrenada. En este punto la red podrá ser usada y los pesos no cambiarán.

Hay que observar que los pasos segundo y tercero del algoritmo anterior constituyen un barrido hacia delante, en el cual las señales se propagan desde la entrada de la red

hacia su salida. Los dos últimos pasos, en cambio, son un barrido hacia detrás, aquí el error calculado se propaga desde la capa final hasta la inicial, a través de la red, para ajustar los pesos.

Como inicialmente indicó Werbos y luego se explicitó en el octavo capítulo del volumen 1 del libro *Parallel Distributed Processing* [Rume 86], la Backpropagation es una formulación compacta del algoritmo del gradiente descendente para el caso de una red multicapa de alimentación hacia adelante totalmente interconectada. El gradiente descendente es una de las soluciones al problema de minimización sin restricciones. En el caso de las redes neuronales el vector de diseño es el vector de pesos de la red

$$\underline{W}^{\mathrm{T}} = (w_1, w_2, ..., w_n) \tag{2.6}$$

el cual no se encuentra sometido a ninguna restricción y ha de minimizar nuestra función de error *J*. Esta función de error se suele tomar como el cuadrado de la distancia entre el vector de salida de la red y el vector objetivo, para todo el conjunto de entrenamiento

$$J = \|\underline{d} - \underline{r}\|^2 = \sum_{i=1}^{m} (d_i - r_i)^2 \tag{2.7}$$

donde $\underline{d}$ es el vector de salida deseada; $\underline{r}$ es la salida de la red cuando se le presenta el vector de entrada del par de entrenamiento correspondiente; y $m=n_L$ es la dimensión del vector de salida.

2.5.2 Minimización sin Restricciones

Para que un punto $\underline{W}^*$ sea un mínimo relativo de $J(\underline{W})$ se ha de cumplir la condición

$$\frac{\partial J}{\partial w_i}(\underline{W} = \underline{W}^*) = 0 \qquad i = 1, 2, ..., n \tag{2.8}$$

y se garantiza un mínimo absoluto si la matriz Hessiana es definida positiva, es decir

$$\left[\frac{\partial^2 J}{\partial w_i \partial w_j}(\underline{W}^*) \right] = \text{ definida positiva} \tag{2.9}$$

Las ecuaciones (2.8) y (2.9) pueden ser utilizadas para identificar los puntos óptimos durante los cálculos numéricos.

Están disponibles varios métodos para resolver un problema de minimización sin restricciones. Estos métodos pueden ser clasificados en dos grandes categorías: búsqueda directa y métodos descendentes. Todos son iterativos por naturaleza, es decir,

comienzan desde una solución inicial de prueba y continúan hacia el punto mínimo de una manera secuencial. El algoritmo general de estos métodos es entonces:

```
Empezar por un punto de prueba W_1
Hacer i=0
Encontrar J(W_1)
Repetir
    i=i+1
    Generar nuevo punto W_{i+1}
    Encontrar J(W_{i+1})
Hasta que se satisfaga la convergencia
```

Es importante hacer notar que todos los métodos de minimización sin restricciones requieren un punto inicial para empezar el procedimiento iterativo y difieren unos de otros sólo en el método de generar el nuevo punto y en el testeo de la optimalidad de ese nuevo punto.

Los métodos directos se basan únicamente en la evaluación de la función objetivo, el punto actual y quizás en los anteriores, para determinar el nuevo punto. Al no utilizar las derivadas parciales de la función *J* suelen denominarse habitualmente métodos no gradiente. Son más adecuados para sistemas sencillos con un número relativamente pequeño de variables y son, por lo general, menos eficientes que los métodos descendentes.

Las técnicas descendentes requieren, además de la evaluación de la función de costo, el cálculo de las derivadas primeras, y posiblemente superiores, para la determinación del nuevo punto. Ya que se utiliza más información acerca de la función a minimizar (a través del uso de sus derivadas), estos métodos, comparados con los de búsqueda directa, son, en general, más eficientes. Los métodos descendentes son también conocidos como métodos del gradiente.

2.5.3 Métodos Descendentes

Como hemos dicho a este tipo de métodos pertenece el algoritmo de la BP. Vamos entonces a definir la forma más básica y los refinamientos más comunes [RaoS 84].

2.5.3.1 Gradiente de una Función

Las derivadas parciales de una función *f*, con respecto a cada una de sus *n* variables, son colectivamente denominadas gradiente y se representan como $\underline{\nabla} f$.

$$\underline{\nabla} f^T = \left\{ \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n} \right\} \tag{2.10}$$

El gradiente es un vector de *n* componentes y tiene una propiedad muy importante: para cualquier punto del espacio *n*-dimensional, el valor de la función se incrementa más rápidamente en la dirección del gradiente. Por lo tanto la dirección del gradiente se denomina la de paso ascendente. Desafortunadamente, la dirección del paso ascendente es una propiedad local y no global. En otras palabras, la dirección del paso ascendente varía de punto a punto.

Como el vector gradiente representa la dirección del paso ascendente, el vector gradiente negativo se denomina la dirección de paso descendente. Por lo tanto, cualquier método que haga uso del vector gradiente alcanzará el punto mínimo más rápidamente que aquel que no haga uso de él. Todos los métodos descendentes hacen uso del vector gradiente, de forma directa o indirecta, para encontrar las direcciones de búsqueda.

Asumiendo que la función es diferenciable, el gradiente en un punto puede ser calculado evaluando la fórmula (2.10) para dicho punto. De todas maneras existen tres situaciones en que la evaluación del gradiente posee ciertos problemas:

a) La función es diferenciable en todos los puntos pero el cálculo de las componentes es imposible o impracticable.

b) La expresión de las derivadas parciales se puede obtener pero ellas requieren gran cantidad de tiempo de cálculo para su evaluación.

c) El gradiente no está definido en todos los puntos.

En el primer y segundo caso se puede utilizar alguna fórmula de diferencia finita, como la diferencia adelante

$$\left.\frac{\partial f}{\partial x_i}\right|_{\underline{x}} \cong \frac{f(\underline{x} + \Delta x_i \underline{u}_i) - f(\underline{x})}{\Delta x_i}, \quad i = 1, 2, \ldots, n \tag{2.11}$$

donde $\underline{u}_i$ es el vector unitario en la dirección de la componente *i*-ésima. Esta fórmula requiere una evaluación adicional de la función para el cálculo de cada componente [Smit 88]. Por ello se necesitan *n* evaluaciones adicionales de la función para obtener la aproximación del gradiente.

Mejores resultados da la fórmula de diferencia central

$$\left.\frac{\partial f}{\partial x_i}\right|_{\underline{x}} \cong \frac{f(\underline{x} + \Delta x_i \underline{u}_i) - f(\underline{x} - \Delta x_i \underline{u}_i)}{\Delta x_i}, \quad i = 1, 2, \ldots, n \tag{2.12}$$

pero esta fórmula requiere dos evaluaciones de la función por cada componente. Por ello se necesitan *2n* evaluaciones adicionales de la función para obtener la aproximación del gradiente.

En el tercer caso no se pueden utilizar las fórmulas de diferencia finita porque el gradiente no está definido en todos los puntos.

Como veremos en el apartado 2.5.6 la BP obtiene fórmulas sencillas para obtener este gradiente en el caso de un perceptrón multicapa.

2.5.3.2 Método Básico del Gradiente Descendente

El uso del vector del gradiente negativo fue usado por primera vez por Cauchy en 1847. En este método partimos de un punto inicial y nos movemos iterativamente hacia el punto óptimo de acuerdo con la regla

$$\underline{x}_{i+1} = \underline{x}_i + \lambda_i \underline{S}_i = \underline{x}_i - \lambda_i \underline{\nabla} f_i \tag{2.13}$$

donde λ_i es el paso elegido a lo largo de la dirección de búsqueda $\underline{S}_i$=-$\underline{\nabla f}_i$. El algoritmo de este método será:

```
Empezar por un punto de prueba x_1
Hacer i=0
Repetir
   i=i+1
   Encontrar la dirección de búsqueda S_i=-∇f_i
   Encontrar λ_i
   Hacer x_{i+1}=x_i+λ_i·S_i
Hasta que x_{i+1} sea óptimo
Tomar x_opt=x_{i+1}
```

La determinación del λ_i es un problema difícil para el cual existen varias estrategias, desde tomar un valor constante hasta realizar interpolaciones para aproximar el valor óptimo. Esta última opción la discutiremos en el apartado 2.5.5. Para detener el proceso iterativo podemos tomar uno de los siguientes criterios

$$\left| \frac{f(\underline{x}_{i+1}) - f(\underline{x}_i)}{f(\underline{x}_i)} \right| \le \varepsilon_1 \tag{2.14}$$

$$\left| \frac{\partial f}{\partial x_i} \right| \le \varepsilon_2, \quad i = 1, 2, \ldots, n \tag{2.15}$$

$$\left| \underline{x}_{i+1} - \underline{x}_i \right| \le \varepsilon_3 \tag{2.16}$$

El método del gradiente descendente puede aparecer como la mejor técnica de minimización sin restricciones, ya que cada búsqueda mono direccional comienza en la mejor dirección. Debido al hecho de que la dirección del paso descendente es una propiedad local el método no es realmente efectivo en la mayoría de los problemas. En un problema de dos dimensiones la aplicación del gradiente descendente conduce a un

camino lleno de segmentos perpendiculares y paralelos. Es decir, aparece un camino en *zigzag* que alarga mucho el proceso. En dimensiones mayores puede que el camino no esté hecho de segmentos paralelos y perpendiculares, sino que aparezcan características diferentes. Para funciones con significativa excentricidad el método se establece en un *zigzag n*-dimensional, con lo cual el proceso se hace desesperadamente lento. Por otro lado, si el contorno de la función objetivo no está muy distorsionado, el método convergerá más rápido.

El método del gradiente descendente puede también ser utilizado con una computadora analógica. Las direcciones del gradiente pueden aplicarse con pasos de longitud infinitesimal y esto lleva a un esquema continuo. El camino descendente $\underline{x}(t)$ puede ser considerado como una función del tiempo cuando los pasos de descenso *ds* son tomados como pasos en el dominio temporal. De esta manera, el camino de minimización $\underline{x}(t)$ es la solución de las *n* ecuaciones diferenciales de primer orden

$$\frac{d\,\underline{x}(t)}{dt} = -\nabla f\left[\underline{x}(t)\right] \tag{2.17}$$

que pueden ser resueltas por una computadora analógica. Las condiciones iniciales para la solución de la ecuación (2.17) se toman para que coincidan con el punto inicial $\underline{x}_1$. El punto límite del camino, cuando $t\rightarrow\infty$, corresponde al punto estacionario de $f(\underline{x})$. A pesar de que esta idea parece interesante, no es útil en la práctica debido a la dificultad de los métodos analógicos en los casos de funciones complicadas o de gran número de variables. Se suma a ello la inherente inexactitud de las computadoras analógicas que pueden conducir a resultados erróneos, sobre todo si nos damos cuenta que nuestro interés no es el camino $\underline{x}(t)$ sino su caso límite.

2.5.3.3 Modificaciones del Método

Numerosas modificaciones se han sugerido a lo largo de los años para acelerar la convergencia del método descendente [Flet 87]. Una de ellas está basada en la idea de usar como dirección de búsqueda

$$\underline{S}_i = \underline{x}_i - \underline{x}_{i-2} \tag{2.18}$$

de vez en cuando, en vez de usar siempre la dirección del gradiente negativo $-\nabla f_i$. El beneficio esperado de esta modificación es que, en caso de *zigzag*, la dirección definida por la ecuación (2.18) queda en la dirección general del mínimo y por lo tanto se puede esperar una convergencia más rápida si nos movemos en esas direcciones ocasionalmente.

Otra modificación, que puede ser considerada una extensión de la idea previa, es la denominada método PARTAN (paralelas tangentes) basado en el gradiente. En este método las direcciones de búsqueda se eligen alternativamente como la dirección del

paso descendente y la dirección dada por una ecuación similar a la (2.18). Se ha demostrado que este método es un tipo de gradiente conjugado (que veremos en el próximo apartado) y que es menos eficiente que éste.

2.5.4 Método del Gradiente Conjugado

Las características de convergencia del gradiente descendente pueden ser mejoradas si aplicamos el método del gradiente conjugado. Se ha mostrado que el uso del gradiente conjugado lleva a una minimización que converge cuadráticamente. Esta propiedad es muy útil porque garantiza que minimizará una función cuadrática en *n* pasos o menos. Ya que algunas funciones se pueden aproximar razonablemente bien a una cuádrica, cerca del punto óptimo, cualquier método que converja cuadráticamente es de esperar que encuentre el punto óptimo en un número finito de iteraciones.

2.5.4.1 Principios Teóricos

Este procedimiento crea cada nueva dirección de búsqueda como una combinación de todas las direcciones de búsqueda previas y de la dirección del gradiente recién calculada. Esta nueva dirección se elige de manera que sea conjugada con las anteriores.

Definición. *Se dice que los vectores $\underline{x}_1$ y $\underline{x}_2$ son conjugados con respecto a la matriz* **M** *si se cumple que*

$$\underline{x}_1^{\mathrm{T}}\mathbf{M}\underline{x}_2 = 0 \tag{2.19}$$

El siguiente teorema es importante en el desarrollo del método.

Teorema 2.1 *Supongamos que el punto $\underline{x}_{i+1}$ es alcanzado después de i pasos durante la minimización de una función cuádrica $f(\underline{x})=1/2\ \underline{x}^T\mathbf{A}\underline{x} + \mathbf{B}^T\underline{x} + \mathrm{C}$. Si las direcciones de búsqueda usadas en el proceso de minimización, $\underline{S}_1, \underline{S}_2, \ldots, \underline{S}_i$, son mutuamente conjugadas con respecto a* **A**, *entonces*

$$\underline{S}_k^T \nabla f_{i+1} = 0 \quad \text{para} \quad \mathrm{k} = 1,2,\ldots,\mathrm{i}$$

Prueba: El gradiente de la función *f* evaluado en el punto $\underline{x}_{i+1}$, viene dado por

$$\nabla f_{i+1} = \mathbf{A}\underline{x}_{i+1} + \mathbf{B} \tag{2.20}$$

Como $\underline{x}_{i+1}$ es alcanzado tras *i* pasos de minimización, se puede escribir como

$$\underline{x}_{i+1} = \underline{x}_1 + \lambda_1^*\underline{S}_1 + \lambda_2^*\underline{S}_2 + \cdots + \lambda_k^*\underline{S}_k + \lambda_{k+1}^*\underline{S}_{k+1} + \cdots + \lambda_i^*\underline{S}_i = \underline{x}_{k+1} + \sum_{j=k+1}^{i}\lambda_j^*\underline{S}_j \tag{2.21}$$

donde λ_j^* es el paso de minimización en la dirección de $\underline{S}_j$. A la vista de la ecuación (2.21), la ecuación (2.20) queda

$$\underline{\nabla} f_{i+1} = \mathbf{A}\left(\underline{x}_{k+1} + \sum_{j=k+1}^{i} \lambda_j^* \underline{S}_j \right) + \mathbf{B} = \underline{\nabla} f_{k+1} + \sum_{j=k+1}^{i} \lambda_j^* \mathbf{A}\underline{S}_j \tag{2.22}$$

Multiplicando por la derecha ambos lados de la ecuación (2.22) por $\underline{S}_k^T$, obtenemos

$$\underline{S}_k^T \underline{\nabla} f_{i+1} = \underline{S}_k^T \underline{\nabla} f_{k+1} + \sum_{j=k+1}^{i} \lambda_j^* \underline{S}_k^T \mathbf{A}\underline{S}_j \tag{2.23}$$

El primer sumando del segundo miembro de la ecuación (2.23) es cero, ya que λ_k^* es el paso óptimo en la dirección de $\underline{S}_k$ y por lo tanto el gradiente en la etapa *k+1* será normal a $\underline{S}_k$. El segundo sumando es cero ya que las direcciones de búsqueda $\underline{S}_1,\underline{S}_2,\ldots,\underline{S}_i$, por hipótesis son conjugadas con respecto a **A**. Como esto es válido para *k=1,2,…,i* hemos probado el teorema. ♦

2.5.4.2 Obtención del Nuevo Algoritmo

Consideremos el desarrollo de un nuevo algoritmo modificando el método del paso descendente, aplicado a una función cuádrica $f(\underline{x})=1/2\ \underline{x}^T\mathbf{A}\underline{x}+\mathbf{B}^T\underline{x}+C$, imponiendo la condición de que las sucesivas direcciones sean mutuamente conjugadas.

Sea el punto de partida de la minimización y sea la dirección del paso descendente la primera dirección de búsqueda. Entonces

$$\underline{S}_1 = -\underline{\nabla} f_1 = -\mathbf{A}\underline{x}_1 - \mathbf{B} \tag{2.24}$$

y

$$\underline{x}_2 = \underline{x}_1 + \lambda_1^* \underline{S}_1 \tag{2.25}$$

donde λ_1^* es el paso que minimiza *f* en la dirección de $\underline{S}_1$, por lo tanto

$$\underline{S}_1^T \underline{\nabla} f_2 = 0 \tag{2.26}$$

Esta ecuación se puede expandir como

$$\underline{S}_1^T \left\{ \mathbf{A}(\underline{x}_1 + \lambda_1^* \underline{S}_1) + \mathbf{B} \right\} = 0 \quad \acute{o} \quad \underline{S}_1^T \mathbf{A}\underline{x}_1 + \lambda_1^* \underline{S}_1^T \mathbf{A}\underline{S}_1 + \underline{S}_1^T \mathbf{B} = 0$$

de las cuales podemos despejar

$$\lambda_1^* = \frac{-\underline{S}_1^T (\mathbf{A}\underline{x}_1 + \mathbf{B})}{\underline{S}_1^T \mathbf{A}\underline{S}_1} = \frac{\underline{S}_1^T}{\underline{S}_1^T} \frac{\underline{\nabla} f}{\mathbf{A}\underline{S}_1} \tag{2.27}$$

Ahora expresamos la segunda dirección de búsqueda como la siguiente combinación lineal

$$\underline{S}_2 = -\underline{\nabla} f_2 + \beta_2 \underline{S}_1 \tag{2.28}$$

donde β_2 ha de ser elegida para hacer $\underline{S}_1$ y $\underline{S}_2$ conjugados. Esto requiere que

$$\underline{S}_1^T \mathbf{A} \underline{S}_2 = 0 \tag{2.29}$$

Sustituyendo las expresiones de $\underline{S}_1$ y $\underline{S}_2$ en esta fórmula y operando llegamos a que la forma que ha de tener β_2 es

$$\beta_2 = \frac{\underline{\nabla} f_2^T \underline{\nabla} f_2}{\underline{\nabla} f_1^T \underline{\nabla} f_1} \tag{2.30}$$

A continuación vamos a considerar la tercera dirección de búsqueda como combinación de las dos anteriores y el gradiente

$$\underline{S}_3 = -\underline{\nabla} f_3 + \beta_3 \underline{S}_2 + \delta_3 \underline{S}_1 \tag{2.31}$$

donde β_3 y δ_3 han de ser de tal manera que hagan $\underline{S}_3$ conjugada de $\underline{S}_1$ y $\underline{S}_2$. Primero consideremos

$$\underline{S}_1^T \mathbf{A} \underline{S}_3 = -\underline{S}_1^T \mathbf{A} \underline{\nabla} f_3 + \beta_3 \underline{S}_1^T \mathbf{A} \underline{S}_2 + \delta_3 \underline{S}_1^T \mathbf{A} \underline{S}_1 = 0 \tag{2.32}$$

Como $\underline{S}_1$ y $\underline{S}_2$ son conjugados entre si, el segundo sumando se hace cero y podemos despejar

$$\delta_3 = \frac{\underline{S}_1^T \mathbf{A} \underline{\nabla} f_3}{\underline{S}_1^T \mathbf{A} \underline{S}_1} \tag{2.33}$$

Sustituyendo la expresión de $\underline{S}_1$ y aplicando el teorema **2.1** llegamos a que δ_3 ha de ser cero. Por la ecuación (2.31) queda

$$\underline{S}_3 = -\underline{\nabla} f_3 + \beta_3 \underline{S}_2 \tag{2.34}$$

El valor de β_3 se puede encontrar exigiendo que $\underline{S}_3$ sea conjugado de $\underline{S}_2$ como hicimos para obtener β_2.

Generalizando la ecuación anterior, podemos expresar la dirección de búsqueda en el *i*-ésimo paso como la combinación lineal

$$\underline{S}_i = -\underline{\nabla} f_i + \beta_i \underline{S}_{i-1} \tag{2.35}$$

donde el valor de β_i se puede obtener, haciendo que $\underline{S}_i$ sea conjugado de $\underline{S}_{i-1}$, como

$$\beta_i = \frac{\underline{\nabla} f_i^T \underline{\nabla} f_i}{\underline{\nabla} f_{i-1}^T \underline{\nabla} f_{i-1}} \tag{2.36}$$

Además de hacer que $\underline{S}_{i-1}$ y $\underline{S}_i$ sean conjugados, se puede demostrar que $\underline{S}_i$ será automáticamente conjugado de todas las direcciones previas $\underline{S}_k$, $k=1, 2, \ldots, i-2$.

El uso de las ecuaciones (2.35) y (2.36) fue sugerido por primera vez para la minimización de funciones generales por Fletcher y Reeves. Su algoritmo se puede esquematizar de la siguiente manera:

Comenzar por un punto inicial arbitrario $\underline{x}_1$
Tomar como primera dirección de búsqueda $\underline{S}_1=-\underline{\nabla}f(\underline{x}_1)=-\underline{\nabla}f_1$
Encontrar el paso óptimo λ_1^* en la dirección de $\underline{S}_1$
Encontrar el punto $\underline{x}_2$ de acuerdo con la relación

$$\underline{x}_2 = \underline{x}_1 + \lambda_1^* \underline{S}_1$$

Hacer i=1
Repetir
  Hacer i=i+1
  Encontrar $\underline{\nabla}f_i=\underline{\nabla}f(x_i)$
  hacer

$$\underline{S}_i = -\underline{\nabla} f_i + \frac{\underline{\nabla} f_i^T \underline{\nabla} f_i}{\underline{\nabla} f_{i-1}^T \underline{\nabla} f_{i-1}} \underline{S}_{i-1} = -\underline{\nabla} f_i + \frac{\left|\underline{\nabla} f_i\right|^2}{\left|\underline{\nabla} f_{i-1}\right|^2} \underline{S}_{i-1} \tag{2.37}$$

  Calcular el paso óptimo λ_i^* en la dirección de $\underline{S}_i$.
  Encontrar el nuevo punto

$$\underline{x}_{i+1} = \underline{x}_i + \lambda_i^* \underline{S}_i \tag{2.38}$$

Hasta que el punto $\underline{x}_{i+1}$ sea óptimo

Para poder realizar este algoritmo nos falta conocer la obtención del paso óptimo en una dimensión. Las distintas soluciones a este problema monodimensional las veremos en el apartado 2.5.5.

2.5.4.3 Resultados del Algoritmo

Para una función cuádrica, ya que las direcciones $\underline{S}_i$ usadas en este método son conjugadas, el proceso ha de converger en $m \leq n$ ciclos. Existen teoremas que permiten conocer el valor de m para este tipo de funciones. De cualquier, manera para cuádricas mal condicionadas (cuyo contorno es altamente excéntrico y distorsionado), el método puede requerir mucho más de n ciclos para converger. La razón se encuentra en los errores de redondeo que se van acumulando. Como $\underline{S}_i$ viene dado por la ecuación (2.37), cualquier error resultante de la determinación de λ_i^*, y los errores acumulados en los sucesivos cálculos, se pasan al vector $\underline{S}_i$. Las direcciones de búsqueda se irán contaminando de esos errores progresivamente.

Para funciones no cuádricas el número de iteraciones no es a priori conocido, por lo que surgen varias posibilidades. Por un lado se puede seguir utilizando la ecuación (2.37) para todos los i, o bien reinicializar periódicamente $\underline{S}_i$ a la dirección del paso descendente. En principio $\underline{S}_i$ puede ser *reseteado* cada n iteraciones, es decir

$$\underline{S}_{cn+1} = -\underline{\nabla} f_{cn+1} \quad \text{para} \quad c = 1,2,\ldots$$

en cuyo caso se obtiene un método de tipo cíclico. Una ventaja aparente de esta estrategia es que, si el proceso evoluciona desde una región no cuádrica hacia las proximidades de la solución, donde $f(x)$ se puede aproximar por una cuádrica, el método de reinicializar vuelve a comenzar el proceso en una zona donde se asegura que acabará en n ciclos. En cambio, si no reinicializamos, aun estando en una zona cuádrica, no estamos en las condiciones en que se asegura la convergencia en n ciclos, y por lo tanto el proceso puede durar mucho más. Es de esperar, entonces, que reinicializando cada n ciclos se llegue a la solución más rápidamente que si no lo hacemos. De hecho para grandes problemas, con ciertos tipos de simetrías, puede ser apropiado reinicializar más frecuentemente.

Otra posibilidad es utilizar diferentes fórmulas para sustituir a la (2.35) y (2.36) en el cálculo de $\underline{S}_i$ (que sean equivalentes a éstas para las funciones cuádricas). Una de ellas utiliza la matriz de proyección simétrica $\mathbf{P}_i$ que se actualiza mediante una fórmula recursiva partiendo de $\mathbf{P}_1=\mathbf{I}$. Entonces $\underline{S}_i=-\mathbf{P}_i\underline{\nabla}f(\underline{x}_1)$ para todo $k=1, 2,\ldots$. Como $\mathbf{P}_{cn+1}=0$, $\mathbf{P}_{cn+1}$ debe ser restaurado a la matriz identidad para todo $c= 1, 2, \ldots$. Este método posee la propiedad de descendencia ($\underline{S}_i^T\underline{\nabla}f(\underline{x}_1)\leq 0$) independiente de la función f. Esta propiedad garantiza que la función puede ser reducida, en una minimización en la dirección de búsqueda, con un paso mayor que cero. Sin embargo el utilizar la matriz de proyección simétrica tiene el inconveniente de la necesidad de almacenar y manejar matrices, frente a las fórmulas originales que sólo manejan vectores. Es posible conservar la propiedad de descendencia utilizando todavía la fórmula sencilla (2.36) y calculando

$$\beta_i = -\frac{\underline{\nabla} f_i^T \underline{\nabla} f_i}{\underline{\nabla} f_{i-1}^T S_{i-1}} \tag{2.39}$$

Existen otras muchas variantes, que son equivalentes a la fórmula (2.35) en el caso de funciones cuádricas, pero cuya utilización no produce una mejora substancial. Sin embargo la alternativa

$$\beta_i = \frac{\left(\underline{\nabla} f_i - \underline{\nabla} f_{i-1}\right)^T \underline{\nabla} f_i}{\underline{\nabla} f_{i-1}^T \underline{\nabla} f_{i-1}} \tag{2.40}$$

propuesta por Polak y Ribiere ha demostrado ser importante.

Cuando se comparan estos métodos con los cuasi-Newton son claramente menos eficientes y menos robustos. Los métodos cuasi-Newton hacen uso de las derivadas parciales de orden superior, por lo que utilizan más información del sistema. En caso de funciones cuádricas se asegura la convergencia en un número mucho menor de iteraciones por lo que deben ser preferidos en circunstancias normales. Su principal inconveniente es que la utilización de derivadas de orden superior hace necesario el

manejo de matrices. Cuando el tamaño del sistema es grande esto hace prohibitiva su utilización. Por otro lado, los métodos del gradiente conjugado no requieren operaciones de matrices. De hecho, el método Fletcher-Reeves precisa sólo de tres vectores *n*-dimensionales para su implementación, y la fórmula Polak-Ribiere necesita cuatro. Por lo tanto los métodos del gradiente conjugado pueden ser los únicos aplicables a grandes problemas. Los casos que vamos a tratar pueden llegar a tener gran cantidad de variables por lo que no consideramos la aplicación de los métodos matriciales.

Para problemas con gran número de variables se ha visto que pueden existir simetrías que lo hagan de una complejidad como un sistema de $m<<n$ variables. Desafortunadamente no suele ser posible comprobar esto a priori. En un caso como éste, y en donde sea adecuado el método del gradiente conjugado, no sería conveniente *resetear* al algoritmo cada *n*, sino cada *m* iteraciones, valor que es difícil de saber. De cualquier manera, si el algoritmo está haciendo pocos progresos entonces $\underline{\nabla} f(\underline{x}_{1+1}) \approx \underline{\nabla} f(\underline{x}_1)$. En este caso, el método Fletcher-Reeves calcula una $\beta_{i+1} \approx 1$, en cambio, la fórmula Polak-Ribiere (2.40) conduce a $\beta_{i+1} \approx 0$ y por lo tanto $\underline{S}_{i+1} \approx -\underline{\nabla} f(\underline{x}_{1+1})$. Esto hace que, cuando no existe evolución, con la fórmula Polak-Ribiere la dirección de búsqueda tienda a reinicializarse de manera automática a la dirección del paso descendente. Parece, entonces, que esta fórmula es la más conveniente en problemas de gran escala.

2.5.5 Búsqueda del Paso Óptimo

Los métodos presentados en apartados anteriores nos servían para determinar la dirección *n*-dimensional en que se realizaría el siguiente desplazamiento del sistema en el proceso de minimización, esto es, la dirección de búsqueda. Un apartado que aún no hemos considerado, en los algoritmos presentados, es el de la determinación del tamaño del desplazamiento (λ_i) en que se moverá el sistema sobre la dirección indicada. En el algoritmo del gradiente descendente, si este desplazamiento es infinitesimal, garantiza la llegada al mínimo, aunque en un tiempo excesivamente largo. En los métodos del gradiente conjugado se supone que este desplazamiento será el óptimo, es decir, el resultante de la minimización unidimensional en la dirección de búsqueda.

Si $f(\underline{x})$ es la función objetivo, el problema de toda minimización unidimensional es encontrar λ^*, el menor valor no negativo de λ, para el cual la función

$$f(\lambda) = f(\underline{x} + \lambda \underline{S}) \tag{2.41}$$

alcance un mínimo local.

Si la función original se puede expresar explícitamente como función de los x_i ($i=1,2, \ldots,n$) podemos escribir las expresiones para cualquier vector $\underline{S}$ y resolviendo

$$\frac{df}{d\lambda}(\lambda) = 0 \tag{2.42}$$

obtendríamos λ^* en términos de $\underline{S}$ y $\underline{x}$. Estaremos aplicando un método analítico.

De cualquier manera, en muchos problemas prácticos la función $f(\lambda)$ no puede expresarse explícitamente en términos de λ, por lo que se ha de recurrir a métodos numéricos. Dentro de los métodos numéricos están los métodos de eliminación y los de interpolación. En ambos casos, una serie de evaluaciones de la función objetivo para distintos valores de la variable de decisión conducen a la obtención del valor óptimo.

Los métodos de interpolación conllevan la aproximación, mediante polinomios de distinto orden, de la función dada. El método de interpolación cuadrática aproxima la función mediante un polinomio de segundo grado. La interpolación cúbica, en cambio, utiliza un polinomio de tercer grado para aproximar la función $f(\lambda)$. El último es un método más exacto pero hace uso de las derivadas de dicha función en cada punto. Cuando la minimización monodimensional proviene de un problema multidimensional no se suele disponer de expresiones analíticas de estas derivadas, por lo que se convierte en un método muy costoso. Por esta razón nos limitaremos a estudiar la interpolación cuadrática.

2.5.5.1 Interpolación Cuadrática

Este método consiste en hacer una aproximación de $f(\lambda)$ mediante una función cuadrática $h(\lambda)$ y encontrar el mínimo $\tilde{\lambda}^*$ de $h(\lambda)$. Sea pues

$$h(\lambda) = a + b\lambda + c\lambda^2 \tag{2.43}$$

la función cuadrática utilizada para aproximar la función $f(\lambda)$. La condición necesaria para el mínimo de $h(\lambda)$ es que

$$\frac{dh}{d\lambda} = b + 2c\lambda = 0 \tag{2.44}$$

y por lo tanto

$$\tilde{\lambda}^* = -\frac{b}{2c} \tag{2.45}$$

La condición suficiente para el mínimo de $h(\lambda)$ es

$$\left.\frac{d^2h}{d\lambda^2}\right|_{\tilde{\lambda}^*} > 0 \tag{2.46}$$

lo que significa que ha de ser

$$c > 0 \tag{2.47}$$

Para despejar las constantes *a*, *b* y *c* de la ecuación (2.43), debemos evaluar la función $f(\lambda)$ en tres puntos. Sea $\lambda=A$, $\lambda=B$ y $\lambda=C$ los puntos elegidos y f_A, f_B y f_c los valores correspondientes de la función $f(\lambda)$. Igualando las dos funciones en dichos puntos tenemos

$$\begin{cases} f_A = h(A) = a + bA + cA^2 \\ f_B = h(B) = a + bB + cB^2 \\ f_C = h(C) = a + bC + cC^2 \end{cases} \tag{2.48}$$

de las cuales se pueden obtener el valor de *a*, *b* y *c*. Sustituyendo sus valores en la expresión del mínimo de $h(\lambda)$ se obtiene que

$$\tilde{\lambda}^* = -\frac{b}{2c} = \frac{(B^2 - C^2) f_A + (A^2 - C^2) f_B + (A^2 - B^2) f_C}{2\left[(B - C) f_A + (A - C) f_B + (A - B) f_C\right]} \tag{2.49}$$

Si los puntos *A*, *B* y *C* se toman como *0*, *t* y *2t* respectivamente, donde *t* es un paso de prueba preseleccionado, la ecuación (2.49) queda

$$\tilde{\lambda}^* = -\frac{b}{2c} = \frac{4 f_B - 3 f_A - f_C}{4 f_B - 2 f_A - 2 f_C} t \tag{2.50}$$

y la condición (2.47) se satisface si

$$\frac{f_A + f_C}{2} > f_B \tag{2.51}$$

que significa que f_B debe ser menor que el valor medio de f_A y f_C. Esto se satisface si f_B está por debajo de la línea que une f_A y f_C.

Este procedimiento completo se aplica en tres etapas:

Etapa 1: En este paso, que sólo es necesario si el problema de minimización monodimensional surge de otro problema de minimización multidimensional, se realiza la normalización del vector $\underline{S}$ dividiendo cada una de sus componentes por

$$\Delta = \max_i \left| s_i \right| \tag{2.52}$$

donde s_i es la componente *i*-ésima de $\underline{S}$. Otra posibilidad es elegir

$$\Delta = (s_1^{\,2} + s_2^{\,2} + \cdots + s_n^{\,2})^{1/2} \tag{2.53}$$

Etapa 2: En esta etapa se eligen los puntos *A*, *B* y *C* más convenientes y se hace una primera aproximación. Por sencillez se toman *A*, *B* y *C* como *0*, *t* y *2t* respectivamente. De esta manera evitamos una evaluación de la función ya que $f_A=f(\lambda=0)$ es generalmente conocido de iteraciones previas. El problema se

reduce a elegir t de manera que se cumpla esta condición (2.51) y se asegure que el mínimo se encuentre en el intervalo $[0,2t]$. El algoritmo será:

Hacer $f_A=f(\lambda=0)$, $t=t_o$, $f_1=f(\lambda=t)$
Si $f_1>f_A$ **entonces**
  Repetir
    Hacer $f_C=f_1$, $t=t/2$, $f_1=f(\lambda=t)$
  Hasta que $f_1<f_A$
  Hacer $f_B=f_1$
Caso Contrario
  Hacer $f_B=f_1$, $f_2=f(\lambda=2t)$
  Mientras $f_2<f_B$
    Hacer $t=2t$, $f_B=f_2$, $f_2=f(\lambda=2t)$
  Hacer $f_C=f_2$

Terminado este proceso tenemos definidos t, f_A, f_B y f_C y podemos calcular $\tilde{\lambda}^*$ mediante (2.50).

Etapa 3: En esta etapa se hacen sucesivas aproximaciones para $\tilde{\lambda}^*$ hasta que sea suficientemente buena. El $\tilde{\lambda}^*$ encontrado en la etapa 2 es el mínimo de la función cuadrática $h(\lambda)$ y nos tenemos que asegurar que está lo suficientemente cerca del mínimo real de $f(\lambda)$ antes de tomar $\lambda^* \approx \tilde{\lambda}^*$. Un test posible es comparar $f(\tilde{\lambda}^*)$ con $h(\tilde{\lambda}^*)$ y considerar a $\tilde{\lambda}^*$ una buena aproximación si no difieren más de una pequeña cantidad. Otra posibilidad es comprobar que $df/d\lambda$ está cerca de cero en $\tilde{\lambda}^*$.

Si los criterios de convergencia no se satisfacen, una nueva función cuadrática

$$h'(\lambda) = a' + b'\lambda + c'\lambda^2 \tag{2.54}$$

se usa para aproximar la función $f(\lambda)$. Para evaluar los valores de los parámetros de esta nueva función utilizaremos los tres mejores valores que poseemos. Este proceso de ajustar otro polinomio para obtener una mejor aproximación de λ^* se conoce como *refitting*.

Consideraremos todas las situaciones y elegiremos los tres mejores puntos de los presentes: A, B, C y $\tilde{\lambda}^*$. Existen cuatro posibilidades, y los tres mejores puntos en cada caso se muestran en la tabla 2.1. Un nuevo valor de $\tilde{\lambda}^*$ se obtiene usando la fórmula general (2.49). Si este $\tilde{\lambda}^*$ tampoco satisface los criterios de convergencia, se reajusta una nueva función cuadrática en base a los criterios de la tabla 2.1.

Si no se desea excesiva exactitud se puede optar por no realizar la tercera etapa o limitar el número de veces que se realiza.

Caso	Características	Puntos para el *refitting* Nuevos		Anteriores
1	$\tilde{\lambda}^*>B$	A	=	B
	$f(\tilde{\lambda}^*)<f_B$	B	=	$\tilde{\lambda}^*$
		C	=	C
		Despreciamos anterior A		
2	$\tilde{\lambda}^*>B$	A	=	A
	$f(\tilde{\lambda}^*)>f_B$	B	=	B
		C	=	$\tilde{\lambda}^*$
		Despreciamos anterior C		
3	$\tilde{\lambda}^*<B$	A	=	A
	$f(\tilde{\lambda}^*)<f_B$	B	=	$\tilde{\lambda}^*$
		C	=	B
		Despreciamos anterior C		
4	$\tilde{\lambda}^*<B$	A	=	$\tilde{\lambda}^*$
	$f(\tilde{\lambda}^*)>f_B$	B	=	B
		C	=	C
		Despreciamos anterior A		

Tabla 2.1 Posibilidades al hacer *refitting*.

2.5.6 Fórmulas de la BP

Como indicamos previamente, la BP es la forma de obtener las derivadas que constituyen el gradiente de la función de error con respecto a los pesos para el caso de una red multicapa de alimentación hacia adelante totalmente interconectada. Vamos a obtener a continuación dichas fórmulas, independientemente de la expresión de la función de activación y de la expresión del error. Debemos llegar a la forma de las derivadas parciales de J con respecto a los distintos tipos de pesos, las cuales constituyen las componentes del vector gradiente. El desarrollo se hará aplicando la regla de la cadena hasta llegar a expresiones en que aparezcan solamente derivadas del error con respecto a las salidas de la red. Para estas derivadas tendremos su forma explícita ya que disponemos de la fórmula analítica que las relaciona.

Para facilitar la forma de las expresiones y hacerlas independientes del número de capas definimos

$$\delta_q(s) \equiv \frac{\partial J}{\partial z_q(s)} \tag{2.55}$$

para cada neurona de cada capa. Ahora, teniendo en cuenta además la fórmula (2.5), la expresión de la derivada parcial de *J* con respecto a cada peso se podrá poner como

$$\frac{\partial J}{\partial w_{pq}(s)} = \frac{\partial J}{\partial z_q(s)} \frac{\partial z_q(s)}{\partial w_{pq}(s)} = \delta_q(s) y_p(s-1) \tag{2.56}$$

es decir, será el producto de la δ de su neurona por el valor de la entrada correspondiente. En el caso del peso de sesgo esta entrada tendrá siempre valor 1.

Estudiemos ahora la forma de δ para distintas capas. Para las capas ocultas (no de salida) tenemos que

$$\delta_q(s) = \frac{\partial J}{\partial z_q(s)} = \frac{\partial J}{\partial y_q(s)} \frac{\partial y_q(s)}{\partial z_q(s)} = \left[\sum_{h=1}^{n_{s+1}} \delta_h(s+1) w_{qh}(s+1) \right] g'(z_q(s)) \tag{2.57}$$

con $s=1,..,L-1$, donde hemos usado que

$$\frac{\partial J}{\partial y_q(s)} = \sum_{h=1}^{n_{s+1}} \frac{\partial J}{\partial z_h(s+1)} \frac{\partial z_h(s+1)}{\partial y_q(s)} = \sum_{h=1}^{n_{s+1}} \delta_h(s+1) w_{qh}(s+1) \tag{2.58}$$

Para la capa de salida

$$\delta_q(L) = \frac{\partial J}{\partial z_q(L)} = \frac{\partial J}{\partial y_q(L)} \frac{\partial y_q(L)}{\partial z_q(L)} = \left[\frac{\partial J}{\partial r_q} \right] g'(z_q(L)) \tag{2.59}$$

Como se observa, se llega a expresiones recursivas que obtienen el parámetro δ de una capa en función de los δ de la capa siguiente, hasta que se llega al caso base, la capa de salida, en que se obtienen los deltas a partir de la derivada de la expresión del error. El proceso de obtención del gradiente entonces será:

```
Seleccionar un par de entrenamiento
Aplicar el vector de entrada y calcular la salida
Para q=1 hasta n_L
   Obtener δ_q(L) con (2.59)
Para s=L-1 hasta 1
   Para q=1 hasta n_s
      Obtener δ_q(s) con (2.57)
Obtener la derivada de J con respecto a cada peso
con (2.56)
```

En el caso de que se elija como función de activación la función sigmoide (2.2) y la función de error (2.7) las expresiones (2.57) y (2.59) quedarán:

$$\delta_q(s) = \left[\sum_{h=1}^{n_{s+1}} \delta_h(s+1) w_{qh}(s+1)\right] y_q(s)\left(1 - y_q(s)\right) \tag{2.60}$$

$$\delta_q(L) = \left[2(d_q - r_q)(-r_q)\right] y_q(s)\left(1 - y_q(s)\right) \tag{2.61}$$

Una vez obtenido el gradiente se aplicará la modificación de los pesos según el gradiente descendente o alguna de sus variantes. Estos métodos se resumen en la fórmula de corrección que se suele asociar a la BP:

$$\underline{w}(k+1) = \underline{w}(k) - \eta(k)\nabla_{\underline{w}} J + \alpha(k)\left[\underline{w}(k) - \underline{w}(k-1)\right] \tag{2.62}$$

donde el índice *k* indica el paso de entrenamiento. Hay que hacer notar que la diferencia $\underline{w}(k)$-$\underline{w}(k-1)$ representa la dirección de modificación en el paso previo de entrenamiento. Por este motivo, eligiendo de forma adecuada la variación de los parámetros escalares $\eta(k)$ y $\alpha(k)$, con esta fórmula se pueden implementar las variantes del gradiente descendente. Por ejemplo, si hacemos $\alpha(k)=0$ y tomamos $\eta(k)$ infinitesimal estaremos en el gradiente descendente básico. En cambio si $\alpha(k)/\eta(k)$ se toma como el resultado de la ecuación (2.36), o de alguna de sus variantes, y $\eta(k)$ se obtiene tras una minimización en la dirección resultante, estaremos frente al gradiente conjugado completo. Entre esos dos extremos se sitúa la opción más habitual en la literatura que es tomar para estos parámetros valores constantes. Para $\alpha(k)$ se suele recomendar un valor entre *0.8* y *0.95*, mientras que el valor para $\eta(k)$ es muy dependiente del problema.

2.5.6.1 Problemas de la BP

El método de entrenamiento que representa la BP no conduce siempre a la solución deseada. Aun en el caso de que este método pueda entrenar la red para el problema propuesto, existen numerosos factores que pueden hacer que el entrenamiento no llegue a buen término.

Un problema inherente a la BP, por tratarse de un método descendente, es la posibilidad de que el proceso quede detenido en un mínimo local. En dicho punto todas las direcciones son de aumento del error y el sistema no evoluciona. Una forma de asegurarnos que no llegamos a un mínimo local es repetir el entrenamiento desde varios conjuntos de pesos iniciales. Si en todos los casos llegamos a ese resultado, o resultados peores, podremos afirmar que se trata del mínimo global. Por otro lado, si estamos en un mínimo local y queremos salir de la situación, existen varios métodos heurísticos. El más utilizado es provocar una ligera modificación aleatoria de todos los pesos que haga salir al sistema de la situación. De cualquier manera, sólo la aplicación de métodos estadísticos, como el presentado en la siguiente sección, son capaces de asegurar la llegada a un mínimo global.

Otro problema muy importante en la BP es la saturación. Si, por el excesivo valor de alguno de los pesos de una neurona, el valor de la suma ponderada *z* se hace grande (en valor absoluto) estaremos en un punto extremo de la función de activación. En este punto la derivada de dicha función se hace prácticamente *0*. Como el valor de la derivada es un factor en todas las fórmulas de propagación del error, ocurrirá que éste no se propagará. El valor de δ para esa neurona será *0* y sus pesos no se corregirán. De forma similar la aportación de esa neurona a los δ de la capa previa también será nula. Si esta situación se da en varias neuronas de una misma capa se puede llegar a detener completamente el proceso. A esta situación se llega por un paso de entrenamiento η demasiado grande. La mejor forma de detectar esa situación es estudiar el valor de *z* de las neuronas y así determinar la que se encuentra saturada. Para salir de este estado, como en el caso anterior, se han ideado métodos heurísticos, como el escalado de los pesos, pero que pueden llevar a la pérdida del entrenamiento realizado.

Otro problema que puede surgir es la inestabilidad de la red. Si ésta se entrena muy rápidamente tenderá a adaptarse al último par de entrenamiento presentado, olvidándose de los anteriores. De esta manera en cada paso de entrenamiento se pierde lo conseguido en el anterior. La forma de evitar esto es realizar un promedio de las correcciones de todos los pares antes de aplicarla a los pesos. De esta manera la corrección será en promedio y se tenderá a la solución óptima global.

2.6 Métodos Estadísticos de Entrenamiento

La principal ventaja de los métodos estadísticos es la posibilidad de evitar los mínimos locales. Comenzaremos presentando los métodos estadísticos generales para, terminar con el del enfriamiento progresivo, uno de los más utilizados [Barn 92].

2.6.1 Métodos Estadísticos Básicos

Se denominan métodos estadísticos a aquellos que hacen uso de números aleatorios para encontrar el mínimo. Veamos a continuación la forma de estos algoritmos.

2.6.1.1 Salto Aleatorio

Es el método estadístico más sencillo. Si consideramos el problema de encontrar el mínimo de $f(\underline{x})$ en un hipercubo *n*-dimensional definido por

$$l_i \le x_i \le u_i, \quad \text{i} = 1,\ 2,\ \ldots,\ \text{n} \tag{2.63}$$

donde l_i y u_i son los limites inferiores y superiores de la variable x_i; el método de salto aleatorio consiste en generar un conjunto de *n* números aleatorios (r_1, r_2,…,r_n) uniformemente distribuidos entre 0 y 1. Cada conjunto de esos números se usa para

encontrar un punto $\underline{x}$, dentro del hipercubo definido por la ecuación anterior, de la siguiente manera

$$\underline{x} = (x_1,\ldots,x_n)^{\mathrm{T}} = (l_1 + r_1(u_1 - l_1),\ldots,l_n + r_n(u_n - l_n))^{\mathrm{T}} \tag{2.64}$$

y evaluar la función en dicho punto. Generando muchos puntos, y el valor de la función en dichos puntos, podemos tomar el menor valor de $f(\underline{x})$ como el punto mínimo deseado.

Este método es muy sencillo y no práctico para problemas de cierta dimensión. Un método más eficiente es conocido como camino aleatorio que se describe a continuación.

2.6.1.2 Método del Camino Aleatorio

Este método consiste en generar una secuencia de aproximaciones al mínimo, cada una basada en la aproximación anterior. Por ello, si $\underline{x}_i$ es la aproximación al mínimo obtenida en la etapa *i-1* la nueva aproximación en la etapa *i*-ésima se obtiene de la relación

$$\underline{x}_{i+1} = \underline{x}_i + \lambda \underline{u}_i \tag{2.65}$$

donde λ es la longitud escalar del paso preestablecida, y $\underline{u}_i$ es un vector unitario aleatorio generado en la etapa *i*-ésima.

El procedimiento de este método se detalla en los siguientes pasos:

```
Elegir punto inicial  x_1 y el paso λ
Hacer f_1=f(x_1) y i=0
Repetir
   Hacer i=i+1
   Generar n números aleatorios y formar u
   Hacer  f_2=f(x_1+λu)
   Si f_1 >f_2 entonces
      Hacer x_1=x_1+λu y f_1 =f_2
```

Este método tiende a minimizar la función objetivo pero puede quedar trabado en una solución pobre. En la figura 2.4 se muestra como esto puede ocurrir en un sistema monodimensional. Asumamos que el punto inicial está situado en **A**. Si el paso escalar es pequeño, todas las desviaciones desde el punto **A** incrementarán la función objetivo y serán rechazados; el punto mejor (**B**) nunca será encontrado y el sistema quedará atrapado en un mínimo local en vez de alcanzar el mínimo global. Si el paso escalar es demasiado grande los entornos de **A** y **B** serán visitados frecuentemente, pero el sistema nunca se establecerá en el mínimo deseado.

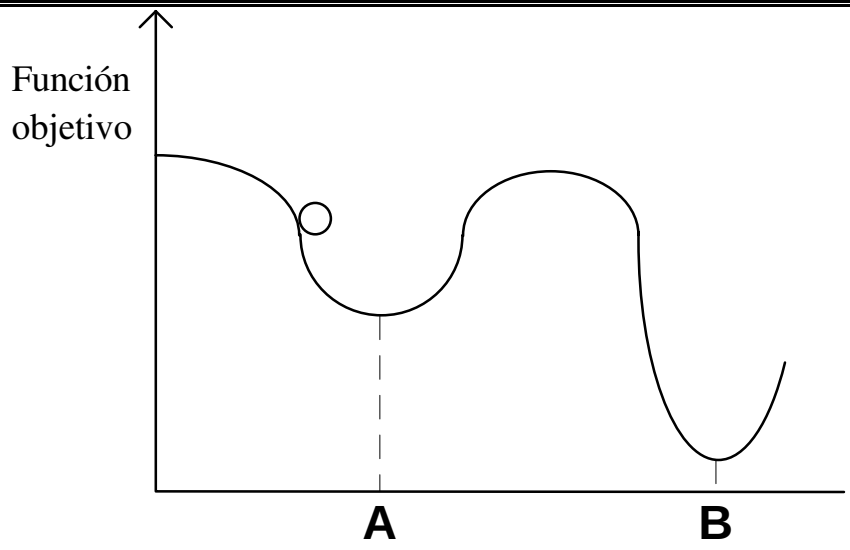

Figura 2.4 - Problema del mínimo local

Una estrategia útil para evitar este problema es empezar con grandes pasos y progresivamente ir reduciendo la longitud del paso. Esto permite que el algoritmo escape de los mínimos locales a la vez que asegura su convergencia. Nos bastaría, en el algoritmo anterior, empezar con un paso escalar suficientemente grande con respecto a la precisión que queramos al final y añadir dos etapas:

```
Elegir punto inicial  x1 y el paso λ grande
Hacer f1=f(x1) y i=0
Repetir
   Hacer i=i+1
   Generar n números aleatorios y formar u
   Hacer  f2=f(x1+λu)
   Si f1 >f2 entonces
      Hacer x1=x1+λu y f1 =f2
   Si i es grande entonces
      Hacer i=0
      Reducir λ
Hasta que λ sea pequeño y no se obtenga mejoría
```

En este nuevo algoritmo, además, se expresa la condición de terminación. Ésta consiste en que el paso sea pequeño y que, a pesar de ello, no exista mejoría en un cierto número de intentos.

2.6.2 Método del enfriamiento progresivo

Es una refinación de los métodos previos y cuya aplicación a las redes neuronales es factible [Wass 89].

2.6.2.1 Introducción

Supongamos que tenemos una bola en una caja de fondo rugoso, del que la figura 2.4 podría ser un corte, y queremos situarla en un pozo más profundo. Un procedimiento es comenzar agitando la caja violentamente. La bola se moverá de un lado a otro, sin asentarse en ningún punto, y en cada instante podrá estar en cualquier punto de la superficie con igual probabilidad. Si la violencia de la agitación se va reduciendo progresivamente, se alcanzará una situación en que la bola estará en los puntos **A** y **B** por cortos periodos de tiempo. Si la agitación continúa reduciéndose, se alcanzará un punto crítico en que la agitación es lo suficientemente fuerte para mover la bola de **A** a **B**, pero no lo suficiente para permitir a la bola subir la cuesta desde **B** hasta **A**. Por lo tanto, al final, la bola quedará en el mínimo global cuando la amplitud de la agitación se reduzca a cero.

Este procedimiento implica, no sólo una dirección de búsqueda aleatoria, sino una longitud de paso también aleatoria, frente a los métodos vistos previamente que poseen paso fijo.

Este procedimiento tiene un gran parecido al proceso de temple de los metales. En un metal, calentado por encima de su punto de fusión, los átomos están en violento movimiento aleatorio. Como todos los sistemas físicos, los átomos tienden al estado de mínima energía (un cristal simple en este caso), pero, a altas temperaturas, la fuerza del movimiento atómico lo evita. A medida que el metal es progresivamente enfriado se asumen estados de energía cada vez menores, hasta que finalmente se alcanza el menor de todos: el mínimo global. En este proceso la distribución de estados de energía está determinada por la relación siguiente:

$$P(e) \propto exp(-e / kT) \tag{2.66}$$

donde:

$P(e)$ es la probabilidad de que el sistema esté en un estado de energía e,

k es la constante de Boltzman,

T es la temperatura en grados Kelvin.

A altas temperaturas el denominador del exponente es muy grande con respecto a cualquier valor de la energía e, por ello el exponente completo estará próximo a cero en todos los casos y $P(e)$ será similar para todos los estados de energía, es decir, los estados de alta energía son tan probables como los de baja. Cuando la temperatura disminuye el exponente será mayor para los valores grandes de e, como éste es negativo la

probabilidad de estados de alta energía se reduce comparada con la probabilidad de estados de baja energía. Cuando la temperatura tiende a cero es muy difícil que el sistema permanezca en un estado de alta energía.

2.6.2.2 Algoritmo de Boltzman

Consiste en la aplicación de la idea que acabamos de presentar a un problema de minimización multidimensional. Los pasos a seguir serán:

Elegir $\underline{x}_1$ y T
Repetir
  Calcular un vector $\underline{u}$ y un paso λ aleatorios
  Hacer $f_2=f(\underline{x}_1+\lambda\underline{u})$
  Si $f_2<f_1$ **entonces**
    Hacer $\underline{x}_1=\underline{x}_1+\lambda\underline{u}$ y $f_1=f_2$
  Caso contrario
    Calcular la probabilidad de aceptar el cambio $P(c)$
    Elegir valor aleatorio de distribución uniforme entre 0 y 1
    Si el valor aleatorio es menor que $P(c)$ **entonces**
      Hacer $\underline{x}_1=\underline{x}_1+\lambda\underline{u}$ y $f_1=f_2$
  Reducir la temperatura T
Hasta que T sea suficientemente pequeño

La probabilidad de aceptar ese cambio se calcula mediante la distribución de Boltzman con la fórmula

$$P(c) = exp(-c / kT) \tag{2.67}$$

donde:

$P(c)$ es la probabilidad de un cambio c en la función objetivo,
k es una constante, análoga a la de Boltzman, dependiente del problema,
T es nuestra variable temperatura.

La aplicación de esta probabilidad permite que el sistema dé, ocasionalmente, un paso que empeore la función objetivo. Esto le da la posibilidad para salir de un mínimo local donde los pasos pequeños siempre incrementan la función objetivo.

El paso escalar λ utilizado debe depender de la variable T. Esta dependencia puede ser de varias maneras. Si emulamos el sistema térmico, λ debe seleccionarse de acuerdo a una distribución gausiana

$$\rho(\lambda) = exp(-\lambda^2 / T^2) \tag{2.68}$$

donde $\rho(\lambda)$ es la densidad de probabilidad para la longitud λ, es decir, la probabilidad de que el paso aleatorio tenga una longitud entre λ y $\lambda+d\lambda$. Utilizaremos un método Monte Carlo para obtener el resultado necesario. Éste se realiza en dos etapas:

1- Encontrar la probabilidad $P(\lambda)$ de que el paso esté entre un valor $-\infty$ y λ. Para ello debemos calcular la integral desde $-\infty$ a λ de $\rho(\lambda)$. En este caso $\rho(\lambda)$ no puede ser integrada por métodos ordinarios, y ha de ser integrada numéricamente y sus resultados tabulados como $P(\lambda)$.

2- Seleccionar un número aleatorio de una distribución uniforme sobre el intervalo $(0,P(\infty))$. Usar este valor como valor de $P(\lambda)$ y mirar el correspondiente valor de λ.

Si se utiliza esta distribución, para que se alcance la convergencia al mínimo global, es necesario que la reducción de la temperatura sea proporcional al logaritmo inverso del tiempo. El enfriamiento del sistema se debe expresar entonces como

$$T(t) = T_0 / \log(1+t) \tag{2.69}$$

donde T_0 es la temperatura inicial y t es el tiempo. Por esta necesidad se hacen precisos largos periodos de tiempo para la convergencia, lo que frecuentemente hace impracticable este método.

2.6.2.3 Algoritmo de Cauchy

Este algoritmo consiste en sustituir la distribución de Boltzman por la de Cauchy en el cálculo del paso escalar de búsqueda. Esta distribución tiene unas "colas" más largas y por lo tanto incrementa la posibilidad de pasos escalares largos. De hecho la distribución de Cauchy tiene una invarianza infinita (indefinida). Por este cambio, la máxima velocidad de reducción de la temperatura se hace inversamente lineal, en vez de inversamente logarítmica como en el caso del algoritmo de Boltzman. Esto reduce drásticamente el tiempo de convergencia. Esta relación se puede expresar como

$$T(t) = T_0 / (1+t) \tag{2.70}$$

La distribución de Cauchy es

$$\rho(\lambda) = \frac{T}{T^2 + \lambda^2} \tag{2.71}$$

donde $\rho(\lambda)$ es la densidad de probabilidad de un paso de longitud λ. Ésta puede ser integrada por los métodos usuales dando una expresión

$$\lambda_c = T \tan\left[P(\lambda)\right] \tag{2.72}$$

donde λ_c es la longitud de paso para la cual la probabilidad es $P(\lambda)$. El método Monte Carlo en este caso es muy sencillo. Para encontrar λ en este caso:

1- Seleccionar un numero aleatorio de una distribución uniforme sobre el intervalo abierto $(-\pi/2,\pi/2)$ (necesario para limitar la función tangente).

2- Sustituir $P(\lambda)$ por este valor en la fórmula (2.72) y calcular el paso de búsqueda λ_c usando la temperatura actual.

2.6.2.4 Método del Calor Específico Artificial

A pesar de la mejora del método de Cauchy los tiempos de convergencia son todavía largos. Puede ser usada una técnica, basada en la termodinámica, para acelerar este proceso. Ésta consiste en ajustar la velocidad de reducción de la temperatura de acuerdo con un "calor específico" artificial calculado durante el proceso de entrenamiento.

El calor específico se define como la velocidad de cambio de la temperatura con la energía. Durante el proceso de temple de un metal ocurren cambios de fase que representan estados de energía discretos. En cada cambio de fase se produce un cambio abrupto en el valor del calor específico. Estos cambios son el resultado del establecimiento del sistema en un mínimo local.

Los procesos de optimización pasan por fases análogas. En los límites de un cambio de fase el calor específico artificial puede cambiar abruptamente. Este calor específico artificial se define como el promedio de cambio de la temperatura con la función objetivo. Como en el ejemplo de la pelota en la caja, la violencia de los movimientos iniciales hace el promedio de la función objetivo virtualmente independiente de pequeños cambios en la temperatura, por ello, el calor específico es prácticamente constante. También, a temperaturas muy bajas, el sistema está confinado en un mínimo, por ello, otra vez, el calor específico es prácticamente constante. Claramente, se puede permitir que la temperatura cambie rápidamente en esos rangos, ya que no existe mejora en la función objetivo.

A temperaturas críticas un pequeño decremento de la temperatura causa un gran descenso en el valor promedio de la función objetivo. Volviendo a la analogía de la pelota, a la "temperatura" en que la pelota tiene suficiente energía media para ir de **A** a **B**, pero insuficiente para ir de **B** a **A**, el valor medio de la función objetivo sufre un cambio abrupto. En esos puntos críticos, el algoritmo debe cambiar la temperatura muy lentamente para asegurar que el sistema no queda accidentalmente atrapado en el punto **A** y por lo tanto atrapado en un mínimo local. Las temperaturas críticas pueden ser detectadas como descensos abruptos en el calor específico artificial, esto es, el promedio de la velocidad de cambio de la temperatura con la función objetivo. Una vez que se alcanza una temperatura crítica, las temperaturas próximas a ese valor deben ser atravesadas lentamente, para asegurar la convergencia al mínimo global. En el resto de

temperaturas se puede usar, con seguridad, una velocidad de descenso mayor, produciendo con ello una significativa reducción en el tiempo de convergencia.

2.6.2.5 Método Gradiente descendente y Cauchy

Los métodos de gradiente descendente tienen la ventaja de la búsqueda directa, es decir, los pesos siempre son modificados en la dirección que minimiza la función de error. Aunque los tiempos de convergencia pueden ser largos, son métodos considerablemente más rápidos que los métodos estadísticos, los cuales encuentran el mínimo global pero, como toman el camino incorrecto numerosas veces, necesitan mucho tiempo de entrenamiento.

La combinación de las dos técnicas puede producir buenos resultados. Haciendo el nuevo punto de prueba igual a la suma del calculado mediante el gradiente descendente y el calculado por el algoritmo de Cauchy puede conducir al mínimo global más rápidamente que ambos métodos por separado.

3

Planteamiento del Problema

En este capítulo definiremos los sistemas continuos de control que van a servir de ejemplo para la comprobación de los métodos estudiados en este trabajo. Presentaremos también los tipos de discretización que utilizaremos para obtener la representación discreta de estos sistemas.

3.1 Introducción

Nuestro objetivo es resolver un problema de control óptimo general sobre un sistema MIMO no lineal continuo determinista definido por una ecuación de estados de la forma

$$\dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u}, t) \tag{3.1}$$

donde $\underline{x} \in X \subseteq \mathbb{R}^n$ y $\underline{u} \in U \subset \mathbb{R}^m$.

Los métodos propuestos en este trabajo resuelven el problema dividiéndolo en etapas, por lo que planteamos la función de coste con la forma discreta y general siguiente

$$J = \sum_{i=0}^{N-1} L(\underline{x}_i, \underline{u}_i, T_i, i) + S_N(\underline{x}_N, T_N, N) \tag{3.2}$$

donde denominamos T_j al intervalo de tiempo de la etapa *j*-ésima. Incluimos estos intervalos como argumentos de las funciones de costo ya que, por lo general, la variable tiempo no está relacionada con la etapa y podrá ser objeto de minimización.

Nuestro objetivo es determinar la secuencia de comandos e intervalos de tiempo $\{(\underline{u}_i, T_i)^o, i = 0,..,N-1\}$ que, para cualquier punto inicial $\underline{x}_0 \in A \subset \mathbb{R}^n$, minimice la función de costo *J*. El punto final podrá ser fijo, deberá caer sobre cierto hiperplano o será totalmente libre, aunque, en este último caso, normalmente se pesará, en la etapa final, la distancia a cierto estado o estados finales.

3.2 Discretización del Sistema

Como dijimos en la sección anterior, el problema está planteado para sistemas continuos pero será resuelto por etapas. En cada etapa se aplicará al sistema el vector comando especificado durante el intervalo de tiempo correspondiente. Es decir, se supone que los sistemas están conectados a un retenedor de orden cero de periodo de muestreo variable. Es necesario conocer la relación entre el estado al comienzo de la etapa, los comandos aplicados, el intervalo de tiempo de la etapa y el estado al final de la etapa.

Una relación de este tipo se puede obtener discretizando la ecuación dinámica del sistema. A continuación veremos las alternativas más utilizadas.

3.2.1 Discretización Clásica

La dinámica del sistema viene representada por un modelo de *n* ecuaciones diferenciales de primer orden. Una primera forma de obtener la discretización es aproximar las derivadas primeras por una diferencia.

Si suponemos que el intervalo *T* es suficientemente pequeño con respecto a la velocidad de cambio de la pendiente de *x*, es decir, si la función es monótona en el intervalo *T*, podemos hacer la aproximación

$$\frac{dx(t)}{dt} \approx \frac{x_{k+1} - x_k}{T} \tag{3.3}$$

donde $x_k = x(t)$ y $x_{k+1} = x(t+T)$. Esta aproximación es la denominada diferencia hacia adelante. Con la misma suposición, se puede definir la diferencia hacia atrás

$$\frac{dx(t)}{dt} \approx \frac{x_k - x_{k-1}}{T} \tag{3.4}$$

y la diferencia central

$$\frac{dx(t)}{dt} \approx \frac{x_{k+1} - x_{k-1}}{2T} \tag{3.5}$$

De estas fórmulas, las dos primeras sirven a nuestros propósitos ya que relacionan el estado en una etapa con el estado en la etapa siguiente (hacia delante o hacia atrás), en cambio la tercera relaciona el estado en una etapa con el estado dos etapas después.

Estas aproximaciones no son muy exactas, sobre todo cuando el periodo T no está ajustado a las variaciones de la función x. Esto es debido a que sólo se tienen en cuenta los valores inicial y final del intervalo y no los valores intermedios. Además, esta aproximación, cuando se aplica a la ecuación de estados del sistemas, afecta sólo a un miembro de la igualdad mientras que el otro se evalúa en un extremo, es decir, se hace

$$\frac{\underline{x}_{k+1} - \underline{x}_k}{T_k} = \underline{f}(\underline{x}_k, \underline{u}_k, t_k) \tag{3.6}$$

cuando sería más conveniente tomar información del otro extremo del intervalo que se aproxima.

A pesar de estas inexactitudes, estas discretizaciones tienen la ventaja de la sencillez en las expresiones, que posibilitan un proceso de cálculo más rápido.

3.2.2 Formulación BPF

La formulación en funciones de bloques de pulsos (BPF) consiste en la representación de las variables mediante bloques de pulsos de anchura variable [RaoG 83] [Patr 89].

Sea una variable *y(t)*, entonces escribiremos el desarrollo

$$y(t) \cong \sum_{k=1}^{N} Y_k \, e_k(t) \tag{3.7}$$

donde N es el número de intervalos, $e_k(t)$ es el pulso e Y_k el coeficiente k-ésimo. Se define

$$e_k(t) = \begin{cases} 1 & t_{k-1} \le t < t_k \\ 0 & \text{en otro caso} \end{cases} \tag{3.8}$$

representando un pulso unitario de anchura $T_k = t_k - t_{k-1}$.

Los coeficientes Y_k se eligen de manera que minimicen el error cuadrático medio entre la evolución real de la variable y la dada por la BPF, es decir, minizan el índice

$$error(y) = \sum_{k=1}^{N} \left[Y_k - \frac{1}{\tau_k} \int_{t_{k-1}}^{t_k} y(t)dt \right]^2 \tag{3.9}$$

Según la definición, esta discretización tiene en cuenta todo el intervalo y no sólo el punto inicial o final de éste, como hacen las discretizaciones clásicas.

Si consideramos la función integral de *y(t)*, ésta también se podrá representar mediante un desarrollo BPF

$$I(y)(t) = \int_0^t y(\lambda)d\lambda = \sum_{k=1}^{N} I_k e_k \tag{3.10}$$

Por la forma en que han sido definidos los coeficientes del desarrollo de *y(t)*, la función integral quedará

$$I(y)(t) = \int_0^t y(\lambda)d\lambda = \sum_{k=1}^{N} \left(\sum_{j=1}^{k-1} Y_j T_j + \frac{T_k}{2} Y_k \right) e_k \tag{3.11}$$

y por lo tanto

$$I_k = \sum_{j=1}^{k-1} Y_j T_j + \frac{T_k}{2} Y_k \tag{3.12}$$

El coeficiente de la etapa $k+1$ se podrá descomponer de la siguiente manera:

$$I_{k+1} = \sum_{j=1}^{k} Y_j T_j + \frac{T_{k+1}}{2} Y_{k+1} = \left(\sum_{j=1}^{k-1} Y_j T_j + \frac{T_k}{2} Y_k \right) + \frac{T_k}{2} Y_k + \frac{T_{k+1}}{2} Y_{k+1} \tag{3.13}$$

Reconocemos, dentro del paréntesis, la expresión del coeficiente de la etapa *k*, con lo que obtendremos

$$I_{k+1} - I_k = \frac{1}{2}(T_k Y_k + T_{k+1} Y_{k+1}) \tag{3.14}$$

consiguiendo así una expresión recursiva. Por lo tanto, si tenemos una ecuación diferencial de primer orden de la forma

$$\dot{x} = y \tag{3.15}$$

e integramos ambos miembros, los coeficientes del desarrollo BPF de estas variables estarán relacionados de la siguiente manera

$$X_{k-1} - X_k = \frac{1}{2}(T_k Y_k + T_{k+1} Y_{k+1}) \tag{3.16}$$

Veamos que propiedades tiene este desarrollo para las operaciones básicas. Para la suma tenemos que si

$$x = y_1 + y_2 + \ldots + y_p \tag{3.17}$$

entonces, desarrollando cada una de ellas

$$\sum_{k=1}^{N} X_k e_k = \sum_{k=1}^{N} Y_{1k} e_k + \sum_{k=1}^{N} Y_{2k} e_k + \ldots + \sum_{k=1}^{N} Y_{pk} e_k \tag{3.18}$$

Agrupando en el segundo miembro los sumandos con el mismo e_k y sacando a estos factor común, tendremos

$$\sum_{k=1}^{N} X_k e_k = (Y_{11} + \ldots + Y_{p1}) e_1 + \ldots + (Y_{1N} + \ldots + Y_{pN}) e_N \tag{3.19}$$

Igualando los coeficientes de e_k a ambos lados de la igualdad llegamos a una expresión que relaciona ambos desarrollos:

$$X_k = Y_{1k} + Y_{2k} + \ldots + Y_{pk} \tag{3.20}$$

Para la multiplicación vamos a actuar de forma similar. Si tenemos que

$$x = y_1 \cdot y_2 \cdot \ldots \cdot y_p \tag{3.21}$$

podemos sustituir cada variable por su desarrollo

$$\sum_{k=1}^{N} X_k e_k = \left(\sum_{k_1=1}^{N} Y_{1k_1} e_{k_1} \right) \cdot \left(\sum_{k_2=1}^{N} Y_{2k_2} e_{k_2} \right) \cdot \ldots \cdot \left(\sum_{k_p=1}^{N} Y_{pk_p} e_{k_p} \right) \tag{3.22}$$

Aplicando la propiedad distributiva de la suma con respecto a la multiplicación, el desarrollo quedará

$$\sum_{k=1}^{N} X_k e_k = \sum_{k_1=1}^{N} \sum_{k_2=1}^{N} \ldots \sum_{k_p=1}^{N} \left(Y_{1k_1} e_{k_1} \cdot \ldots \cdot Y_{pk_p} e_{k_p} \right) \tag{3.23}$$

De estos términos cruzados, por la forma en que están definidos los e_k, serán todos 0 salvo aquellos para los que $k_1 = k_2 = \ldots = k_p$, por lo que el segundo miembro queda reducido a

$$\sum_{k=1}^{N} X_k e_k = \sum_{k=1}^{N} (Y_{1k} \cdot Y_{2k} \cdot \ldots \cdot Y_{pk}) e_k \tag{3.24}$$

donde, al ser e_k unitario, $(e_k)^p = e_k$. Igualando, como antes, los coeficientes para cada e_k a ambos lados de la igualdad tendremos que

$$X_k = Y_{1k} \cdot Y_{2k} \cdot \ldots \cdot Y_{pk} \tag{3.25}$$

Con estas propiedades es posible sustituir la suma/producto de funciones por la suma/producto de los coeficientes de su desarrollo. Esto permite aplicar esta formulación

a una gran variedad de funciones y ecuaciones diferenciales, como veremos en nuestros problemas ejemplo.

3.3 Problemas Ejemplos

En los capítulos siguientes, en que se presentan los distintos métodos diseñados para la resolución del problema de control planteado, usaremos una serie de ejemplos para estudiar sus resultados. Estos sistemas ejemplo se han elegido de complejidad creciente para ver las dificultades que surgen con el aumento de ésta. También se han escogido algunos de ellos basados en plantas reales para permitir el estudio posterior sobre una aplicación real.

En esta sección presentaremos las ecuaciones para dichos sistemas e indicaremos el tipo de índice de coste utilizado. Asimismo, veremos la expresión de las distintas discretizaciones.

Si no se indica lo contrario, las magnitudes físicas utilizadas en este y los restantes capítulos estarán expresadas en unidades gramos/centímetros/segundos, según corresponda.

3.3.1 Primer Sistema

El primero de los problemas ejemplo consiste en un sistema de segundo orden no lineal, basado en un problema lineal. El sistema lineal inicial era el representado por una función de transferencia con un polo y un integrador:

$$G(s) = \frac{1}{s(s-a)} \tag{3.26}$$

Una representación en variables de estado puede ser

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = a_1 x_2 + u \end{cases} \tag{3.27}$$

Para este sistema, con índices de costo de tiempo mínimo y tiempo-comando, existen soluciones al PCOG basadas en el principio de mínimo de Pontryagin. Estas soluciones son del tipo de control denominado Bang-Bang [Luqu 83] [Serr 84] [More 85] [More 86]. También ha sido estudiado mediante programación dinámica [Acos 91] [More 92].

Queremos, partiendo de este sistema lineal, añadir términos no lineales pesados por distintos parámetros. El sistema planteado es entonces

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = ax_2 + bx_2x_1^2 + dx_1 + cu \end{cases} \tag{3.28}$$

Además este sistema, dado unos ciertos valores a sus parámetros, se convierte en un conocido sistema no lineal: el sistema de *Van der Pol* [Gibs 63] [Athe 82] [Bill 84] [Cook 86] [Bank 88]:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = a_v x_2(b_v - x_1^2) - x_1 + u \end{cases} \tag{3.29}$$

El valor de los parámetros para los cuales nos encontramos en uno u otro sistema se muestran en la tabla 3.1.

Sistema Lineal	Sist. Intermedio	Sist. *Van der Pol*
a_l	a	$a_v b_v$
0	b	$-a_v$
0	d	-1

Tabla 3.1 Casos especiales de los valores de los parámetros

Este sistema será estudiado fundamentalmente para índices de tiempo mínimo con puntos iniciales sobre el eje x_1 y punto final el origen. Sabemos, del sistema lineal, que las trayectorias óptimas, para estos puntos iniciales, estarán contenidas en el IV cuadrante.

3.3.1.1 Ecuaciones Discretizadas

La expresión discretizada de las ecuaciones (3.28) será, utilizando la aproximación de diferencia adelante,

$$\begin{cases} (x_{1k+1} - x_{1k})/T_k = x_{2k} \\ (x_{2k+1} - x_{2k})/T_k = ax_{2k} + bx_{2k}x_{1k}^{\ 2} + dx_{1k} + cu_k \end{cases} \tag{3.30}$$

y despejando los valores de la etapa $k+1$ tendremos

$$\begin{cases} x_{1k+1} = x_{1k} + T_k x_{2k} \\ x_{2k+1} = x_{2k} + T_k\left[ax_{2k} + bx_{2k}x_{1k}^{\ 2} + dx_{1k} + cu_k\right] \end{cases} \tag{3.31}$$

De forma similar, para el caso de diferencia atrás

$$\begin{cases} (x_{1k} - x_{1k-1})/T_k = x_{2k} \\ (x_{2k} - x_{2k-1})/T_k = ax_{2k} + bx_{2k}x_{1k}^{\ 2} + dx_{1k} + cu_k \end{cases} \tag{3.32}$$

Para aplicar la formulación BPF hacemos uso de las propiedades de la integral, la suma y el producto. Con ello la primera de las ecuaciones en (3.28) quedará

$$x_{1k+1} - x_{1k} = \frac{1}{2}\left[T_k x_{2k} + T_{k+1} x_{2k+1}\right] \tag{3.33}$$

mientras que la segunda será

$$\begin{aligned} x_{2k+1} - x_{2k} = {} & \frac{a}{2}\left[T_k x_{2k} + T_{k+1} x_{2k+1}\right] + \frac{b}{2}\left[T_k x_{2k} x_{1k}^2 + T_{k+1} x_{2k+1} x_{1k+1}^2\right] + \\ & + \frac{d}{2}\left[T_k x_{1k} + T_{k+1} x_{1k+1}\right] + \frac{c}{2}\left[T_k u_k + T_{k+1} u_{k+1}\right] \end{aligned} \tag{3.34}$$

Como se puede observar, las expresiones obtenidas mediante la BPF involucran los valores de las variables tanto en el punto inicial como el final del intervalo por lo que son mucho más exactas, aunque también más engorrosas. Para simplificarlas se suele considerar $T_{k+1} \approx T_k$. Si hacemos esta aproximación las ecuaciones anteriores quedarán.

$$\begin{cases} 2(x_{1k+1} - x_{1k}) = T_k (x_{2k} + x_{2k+1}) \\ 2(x_{2k+1} - x_{2k}) = T_k \left[a(x_{2k} + x_{2k+1}) + b(x_{2k} x_{1k}^2 + x_{2k+1} x_{1k+1}^2) + d(x_{1k} + x_{1k+1}) + c(u_k + u_{k+1})\right] \end{cases} \tag{3.35}$$

3.3.2 Segundo Sistema

El segundo sistema ejemplo objetivo de nuestro estudio es un modelo del robot móvil llamado Hilare [Wals 94] cuyo esquema se muestra en la figura 3.1.

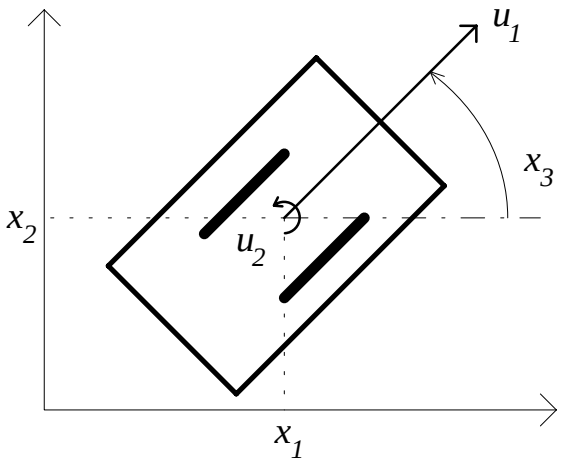

Figura 3.1 - Esquema del robot Hilare

Se trata de un vehículo de dos ruedas que puede girar y avanzar. Tiene tres grados de libertad: su posición sobre el plano y el ángulo de giro; y dos entradas: la velocidad de avance u_1 y la de giro u_2. Se trata pues de un sistema de tercer orden con dos comandos.

Para el modelo tomamos como variables de estado las coordenadas de su centro (x_1,x_2) con respecto a unos ejes cartesianos y el ángulo de su dirección de avance x_3 con respecto al primero de esos ejes. La ecuación de estado del sistema será entonces:

$$\begin{cases} \dot{x}_1 = \cos(x_3)u_1 \\ \dot{x}_2 = \text{sen}(x_3)u_1 \\ \dot{x}_3 = u_2 \end{cases} \tag{3.36}$$

Se trata de un sistema no lineal por las funciones trigonométricas que afectan a la tercera variable de estado.

Este sistema será estudiado fundamentalmente para índices de tiempo mínimo con puntos iniciales sobre el eje x_2 y punto final el origen, restringiendo x_1 a valores positivos y sólo permitiendo que el robot avance, es decir, limitando $u_1 \geq 0$.

Alternativamente, consideraremos un modelo en el cual exista una deriva en la dirección de x_2 quedando ahora las ecuaciones

$$\begin{cases} \dot{x}_1 = \cos(x_3)u_1 \\ \dot{x}_2 = a + \text{sen}(x_3)u_1 \\ \dot{x}_3 = u_2 \end{cases} \tag{3.37}$$

donde a es un parámetro constante pequeño.

3.3.2.1 Ecuaciones Discretizadas

Aplicando diferencia adelante a las ecuaciones de estado tenemos

$$\begin{cases} (x_{1k+1} - x_{1k})/T_k = \cos(x_{3k})u_{1k} \\ (x_{2k+1} - x_{2k})/T_k = \text{sen}(x_{3k})u_{1k} \\ (x_{3k+1} - x_{3k})/T_k = u_{2k} \end{cases} \tag{3.38}$$

de donde podemos despejar

$$\begin{cases} x_{1k+1} = x_{1k} + T_k \cos(x_{3k})u_{1k} \\ x_{2k+1} = x_{2k} + T_k \,\text{sen}(x_{3k})u_{1k} \\ x_{3k+1} = x_{3k} + T_k u_{2k} \end{cases} \tag{3.39}$$

Las expresiones para la diferencia atrás son

$$\begin{cases} (x_{1k} - x_{1k-1})/T_k = \cos(x_{3k})u_{1k} \\ (x_{2k} - x_{2k-1})/T_k = \text{sen}(x_{3k})u_{1k} \\ (x_{3k} - x_{3k-1})/T_k = u_{2k} \end{cases} \tag{3.40}$$

La aplicación de la formulación BPF no es directa sobre las funciones trigonométricas. Para poder hacerlo las sustituimos por un desarrollo en serie equivalente. Tras esta sustitución la primera ecuación de (3.36) quedara

$$\dot{x}_1 = \cos(x_3)u_1 = \left[1 - \frac{x_3^{\,2}}{2!} + \frac{x_3^{\,4}}{4!} - \frac{x_3^{\,6}}{6!} + \cdots\right]u_1 \tag{3.41}$$

Este desarrollo, si utilizamos sus infinitos términos, es valido para cualquier valor de su argumento. Ahora en la ecuación sólo aparecen sumas y productos de variables. Aplicando las propiedades de la BPF, esta ecuación quedará:

$$x_{1k+1} - x_{1k} = \frac{1}{2}\left[(T_k u_{1k} + T_{k+1}u_{1k+1}) - \frac{(T_k u_{1k} x_{3k}^{\,2} + T_{k+1}u_{1k+1}x_{3k+1}^{\,2})}{2!} + \cdots\right] \tag{3.42}$$

Si sacamos factor común Tu_1 de cada sumando y separamos los términos correspondientes a cada etapa tendremos que

$$x_{1k+1} - x_{1k} = \frac{1}{2}T_k u_{1k}\left[1 - \frac{x_{3k}^{\,2}}{2!} + \frac{x_{3k}^{\,4}}{4!} + \cdots\right] + \frac{1}{2}T_{k+1}u_{1k+1}\left[1 - \frac{x_{3k+1}^{\,2}}{2!} + \frac{x_{k+13}^{\,4}}{4!} + \cdots\right] \tag{3.43}$$

Dentro de los corchetes reconocemos la expresión del desarrollo para el coseno, con sus infinitos términos, por lo que podemos simplificar poniendo

$$x_{1k+1} - x_{1k} = \frac{1}{2}T_k u_{1k}\cos(x_{3k}) + \frac{1}{2}T_{k+1}u_{1k+1}\cos(x_{3k+1}) \tag{3.44}$$

Podemos operar de forma similar para la segunda ecuación de (3.36), utilizando un desarrollo similar para el seno. La expresión completa del sistema discretizado mediante BPF será entonces

$$\begin{cases} 2(x_{1k+1} - x_{1k}) = T_k u_{1k}\cos(x_{3k}) + T_{k+1}u_{1k+1}\cos(x_{3k+1}) \\ 2(x_{2k+1} - x_{2k}) = T_k u_{1k}\,\text{sen}(x_{3k}) + T_{k+1}u_{1k+1}\,\text{sen}(x_{3k+1}) \\ 2(x_{3k+1} - x_{3k}) = T_k u_{2k} + T_{k+1}u_{2k+1} \end{cases} \tag{3.45}$$

Si hacemos la aproximación de que $T_{k+1} \approx T_k$, entonces las ecuaciones discretizadas serán

$$\begin{cases} 2(x_{1k+1} - x_{1k}) = T_k\left[u_{1k}\cos(x_{3k}) + u_{1k+1}\cos(x_{3k+1})\right] \\ 2(x_{2k+1} - x_{2k}) = T_k\left[u_{1k}\,\mathrm{sen}(x_{3k}) + u_{1k+1}\,\mathrm{sen}(x_{3k+1})\right] \\ 2(x_{3k+1} - x_{3k}) = T_k\left[u_{2k} + u_{2k+1}\right] \end{cases} \tag{3.46}$$

3.3.3 Tercer Sistema

El tercer sistema está basado en una planta real, disponible como banco de prácticas [Alec 90] [Hern 92], consistente en dos tanques a distinta altura interconectados por debajo, el segundo de los cuales posee una resistencia calefactora. Un esquema de este sistema se presenta en la figura 3.2. Las distintas magnitudes que allí aparecen son:

Parámetros constantes:

- A_1 Área del primer depósito.
- A_2 Área del segundo depósito.
- h_0 Diferencia de altura entre los depósitos.
- T_{atm} Temperatura ambiente.
- r_1 Constante de resistencia del orificio del primer depósito.
- r_2 Constante de resistencia del orificio del segundo depósito.

Variables:

- h_1 Altura del líquido en el primer depósito.
- h_2 Altura del líquido en el segundo depósito.
- q_{ent1} Caudal de entrada de líquido en el primer depósito.

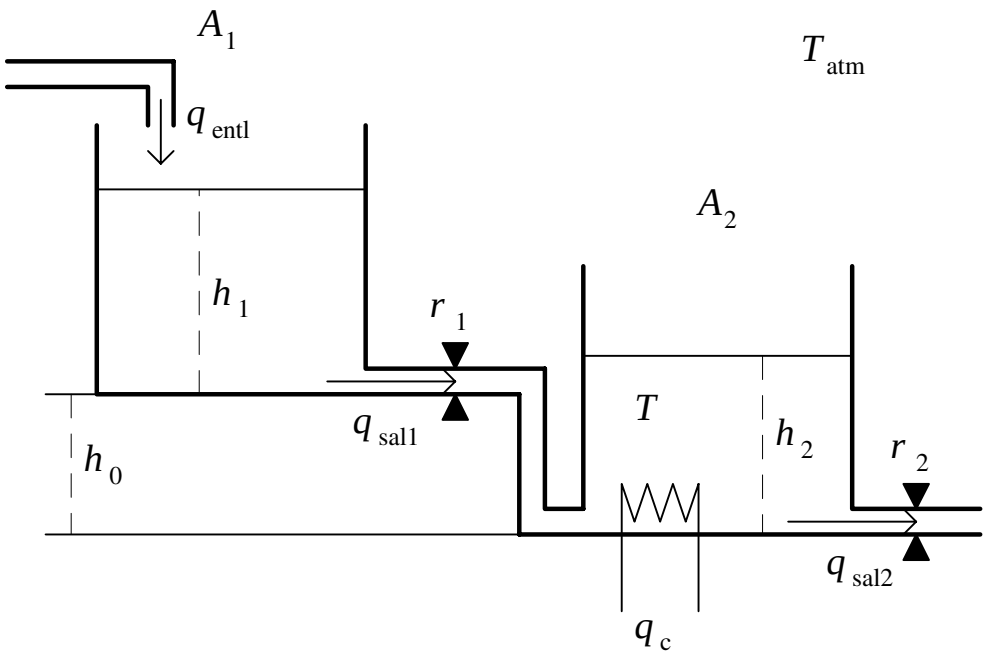

Figura 3.2 - Esquema del Sistema de Tanques Interconectados

q_{sal1} Caudal salida del líquido del primer depósito.

q_{sal2} Caudal salida del líquido del segundo depósito.

T Temperatura del líquido en el segundo depósito.

q_c Energía, en forma de calor, suministrada al segundo depósito.

Para modelar este sistema estudiaremos, en primer lugar, la dinámica de los niveles de los tanques y, en segundo lugar, la variación de la temperatura del segundo tanque.

3.3.3.1 Dinámica de los Niveles

Las ecuaciones que relacionan los caudales de entrada y salida en un depósito, suponiendo que la densidad del líquido no cambia con la temperatura, son las siguientes:

$$\frac{dV}{dt} = \frac{dh}{dt}A = (q_{ent} - q_{sal}) \tag{3.47}$$

Si suponemos un flujo laminar a través del orificio de salida, el caudal se puede modelar como proporcional a la diferencia de altura del líquido a uno y otro lado del orificio según la fórmula

$$q_{sal} = \frac{\Delta h}{r} \tag{3.48}$$

donde r es un parámetro denominado resistencia del orificio y que viene dado por la fórmula

$$r = \sqrt{\frac{2}{A^2 g}\Delta h} \tag{3.49}$$

Como vemos, se trata de un sistema de parámetros distribuidos ya que este parámetro depende de una de las variables. En muchas ocasiones, para poder tratar este problema de forma sencilla, se supone un valor constante para r. Si no hacemos esta aproximación el caudal de salida quedará

$$q_{sal} = \Delta h\sqrt{\frac{A^2 g}{2\Delta h}} = \sqrt{\frac{A^2 g}{2}}\sqrt{\Delta h} \tag{3.50}$$

Sustituyendo esta expresión en la ecuación (3.47) y despejando la variación de la altura, tendremos que

$$\frac{dh}{dt} = \frac{1}{A}\left(q_{ent} - \sqrt{\frac{A^2 g}{2}}\sqrt{\Delta h}\right) = \frac{q_{ent}}{A} - RP(\Delta h) \tag{3.51}$$

donde

$$R = \sqrt{\frac{g}{2}} \tag{3.52}$$

y la función *P(x)* está definida como

$$P(x) = \sqrt{x} \tag{3.53}$$

En caso de que queramos hacer la simplificación antes mencionada haríamos *P(x)=x*.

Aplicando la fórmula (3.51) al tanque 1, y teniendo en cuenta que en nuestra planta $A_1=A_2=A$, tenemos que

$$\frac{dh_1}{dt} = \frac{q_{ent1}}{A} - \frac{q_{sal1}}{A} = q_l - R_1 P(h_1 - h_2 + h_0) \tag{3.54}$$

donde hemos denominado q_l al cociente q_{ent1}/A. Para el tanque 2 tenemos que

$$\frac{dh_2}{dt} = \frac{q_{ent2}}{A} - \frac{q_{sal2}}{A} = \frac{q_{sal1}}{A} - \frac{q_{sal2}}{A} = R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2) \tag{3.55}$$

3.3.3.2 Dinámica de la Temperatura

Para estudiar la dinámica de la temperatura en el segundo tanque detallamos los distintos factores en que se emplea la energía suministrada por la resistencia:

- Energía destinada a incrementar la temperatura del líquido que varía en el segundo tanque, desde la temperatura ambiente a la temperatura *T*

  $$E_1 = mC_{liq}(T - T_{atm}) = \dot{h}_2 \rho_{liq} C_{liq}(T - T_{atm}) \tag{3.56}$$

  donde ρ_{liq} y C_{liq} son, respectivamente, la densidad y el calor especifico del líquido. Sustituyendo la expresión (3.55), tendremos que

  $$E_1 = C_p \left[R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2) \right](T - T_{atm}) \tag{3.57}$$

  donde hemos agrupado $C_p=\rho_{liq}\cdot C_{liq}$.

- Energía que se acumula en el líquido y por lo tanto aumenta su temperatura

  $$E_2 = C_{liq} \frac{dT}{dt} \tag{3.58}$$

- Energía que se acumula en el recipiente

  $$E_3 = C_{alu} \frac{dT}{dt} \tag{3.59}$$

  donde C_{alu} es el calor especifico del recipiente (construido de aluminio).

- Energía que se pierde por las paredes

  $$E_4 = \frac{(T - T_{atm})}{R_T} \tag{3.60}$$

donde R_T es un parámetro para la radiación de energía a través de las paredes.

Una vez definidos estas magnitudes, por la ley de conservación de la energía, podemos poner que

$$q_c = E_1 + E_2 + E_3 + E_4 \tag{3.61}$$

Sustituyendo las expresiones correspondientes, tendremos que

$$q_c = \left[C_p\left(R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2)\right) + \frac{1}{R_T} \right](T - T_{atm}) + \left(C_{liq} + C_{alu}\right)\frac{dT}{dt} \tag{3.62}$$

Definimos la variable θ=T-T_{amb}. Como suponemos que la temperatura ambiente es constante, será cierto que

$$\frac{dT}{dt} = \frac{d\theta}{dt} \tag{3.63}$$

Con esta nueva variable la ecuación (3.62) queda

$$q_c = \theta\left[\frac{1}{R_T} + C_p\left(R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2)\right) \right] + \dot{\theta} C_s \tag{3.64}$$

donde hemos definido C_s=C_{liq}+C_{alu}. Despejando la derivada de θ, la ecuación de estado para la temperatura será

$$\dot{\theta} = \frac{q_c}{C_s} - \frac{\theta}{C_s}\left[\frac{1}{R_T} + C_p\left(R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2)\right) \right] \tag{3.65}$$

El modelo matemático completo del sistema será

$$\begin{cases} \dot{h}_1 = -R_1 P(h_1 - h_2 + h_0) + q_l \\ \dot{h}_2 = R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2) \\ \dot{\theta} = \left\{ \theta\left[-1/R_T - C_p\left(R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2)\right) \right] + q_c \right\} / C_s \end{cases} \tag{3.66}$$

donde las variables de estado son h_1, h_2 y θ, y los comandos q_l y q_c.

Como se ve, se trata de un sistema de tercer orden con dos entradas. Sus ecuaciones son algo más complejas que las de los sistemas anteriores. Una característica interesante es que las dos primeras ecuaciones de estado están desacopladas de la tercera. Es decir, al suponer que el volumen del líquido no varía con la temperatura, como ocurre en la práctica, la variación de ésta no influye para nada en los niveles y caudales. En cambio, la temperatura del segundo depósito depende de la cantidad de líquido que le entra y por lo tanto de las otras dos variables de estado.

Este sistema será estudiado fundamentalmente para índices de tiempo mínimo partiendo de un estado con ambos depósitos vacíos hasta conseguir que el segundo tanque esté suministrando un caudal determinado a una temperatura determinada. Como en los sistemas anteriores se tendrán en cuenta las limitaciones en las variables de estado y sobre todo en los comandos.

3.3.3.3 Ecuaciones Discretizadas

Aplicando a (3.66) la aproximación de diferencia adelante y despejando las variables en la etapa *k+1* tendremos

$$\begin{cases} h_{1k+1} = h_{1k} + T_k\left[-R_1P(\Delta h_k) + q_{lk}\right] \\ h_{2k+1} = h_{2k} + T_k\left[R_1P(\Delta h_k) - R_2P(h_{2k})\right] \\ \theta_{k+1} = \theta_k + T_k\left\{-\theta_k\left[1/R_T + C_p\left(R_1P(\Delta h_k) - R_2P(h_{2k})\right)\right] + q_{ck}\right\}\Big/C_s \end{cases} \tag{3.67}$$

donde denominamos Δh a $h_1 - h_2 + h_0$.

Si aplicamos la aproximación de diferencia atrás:

$$\begin{cases} h_{1k} - h_{1k-1} = T_k\left[-R_1P(\Delta h_k) + q_{lk}\right] \\ h_{2k} - h_{2k-1} = T_k\left[R_1P(\Delta h_k) - R_2P(h_{2k})\right] \\ \theta_k - \theta_{k-1} = T_k\left\{-\theta_k\left[1/R_T + C_p\left(R_1P(\Delta h_k) - R_2P(h_{2k})\right)\right] + q_{ck}\right\}\Big/C_s \end{cases} \tag{3.68}$$

Finalmente, si aplicamos a (3.66) la formulación BPF y aproximamos $T_{k+1} \approx T_k$, cada una de sus ecuaciones se discretizarán de la siguiente manera:

$$2(h_{1k+1} - h_{1k})/T_k = \left[-R_1P(\Delta h_k) + q_{lk}\right] + \left[-R_1P(\Delta h_{k+1}) + q_{lk+1}\right] \tag{3.69}$$

$$2(h_{2k+1} - h_{2k})/T_k = \left[R_1P(\Delta h_k) - R_2P(h_{2k})\right] + \left[R_1P(\Delta h_{k+1}) - R_2P(h_{2k+1})\right] \tag{3.70}$$

$$\begin{aligned} 2(\theta_{k+1} - \theta_k)/T_k = &\left\{-\theta_k\left[1/R_T + C_p\left(R_1P(\Delta h_k) - R_2P(h_{2k})\right)\right] + q_{ck}\right\}\Big/C_s + \\ &+\left\{-\theta_{k+1}\left[1/R_T + C_p\left(R_1P(\Delta h_{k+1}) - R_2P(h_{2k+1})\right)\right] + q_{ck+1}\right\}\Big/C_s \end{aligned} \tag{3.71}$$

donde *P(x)* ha de poder desarrollarse en serie para cualquier valor de su argumento y así poderle aplicar el procedimiento que seguimos en la sección anterior con el seno y el coseno. Esta condición es cierta para las funciones indicadas.

4

Aplicación de la Programación Dinámica

En trabajos previos se había realizado una primera aproximación a la reducción de la complejidad de la PD y su aplicación a los sistemas lineales [Acos 91]. En el presente trabajo los métodos de reducción de la complejidad de la PD se han formalizado y se han encontrado las condiciones necesarias y suficientes para su aplicabilidad [More 94]. Además las hemos utilizado cuando aplicamos la PD a tres sistemas ejemplo. Como vimos en el capítulo anterior se trata de tres sistemas no lineales muy diferentes entre sí que pondrán de manifiesto la puesta en práctica de estas condiciones.

4.1 Mejoras a la Programación Dinámica

4.1.1 Índices de Costo Generales

Como indicamos en el capítulo anterior deseamos resolver problemas de control óptimo con índices generales, es decir, que en la función de costo pueda aparecer cualquier variable, incluida el tiempo. Veremos a continuación las dificultades que esto conlleva y como es posible solventarlas.

4.1.1.1 Propiedades de la Variable de Etapa

Para solucionar mediante la PD problemas generales, como el presentado en el capítulo 3, se debe utilizar el método computacional. Como vimos en el apartado 1.3.4, es necesario elegir una variable de etapa. Ésta ha de ser monótona, es decir, ha de crecer o decrecer independientemente del valor del resto de las variables. De esta manera se puede determinar el concepto de etapa que permitirá producir los dos barridos de la PD.

Para aclarar este problema veamos un ejemplo sencillo. Sea un sistema inercial con las ecuaciones siguientes

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = u \end{cases} \tag{4.1}$$

Si tomamos a x_2 como variable de etapa vemos que ésta crecerá o decrecerá dependiendo del valor que elijamos para el comando u. Si, realizando el primer barrido de la PD, nos encontramos en un estado de la etapa k, cuando probemos valores de u positivos el estado que alcancemos estará en la etapa siguiente $k+1$ (de x_2 mayor). En cambio, si probamos un valor de u negativo el sistema pasará a un valor de x_2 menor que el actual y con ello a la etapa $k-1$, para la cual no tenemos aún definida la función de mínimo coste.

Una variable que es claramente monótona es el tiempo. Sabemos que éste avanza indefectiblemente. Por esta propiedad es utilizado habitualmente como variable de etapa discretizándolo a intervalos fijos o variables.

Otra característica importante de la variable de etapa es que no puede entrar en el índice de calidad. Al estar fijado inicialmente el número total de etapas en que vamos a resolver nuestro problema sabremos, a priori, que intervalo de nuestra variable de etapa vamos a necesitar independientemente de la trayectoria seguida y los comandos aplicados. Por ejemplo si utilizamos el tiempo como variable de etapa con un intervalo T_k, en general diferente para cada etapa, el tiempo total de cualquier trayectoria será

$$T_{total} = \sum_{k=1}^{N} T_k \tag{4.2}$$

sin importar los comandos que apliquemos en dicha trayectoria.

Por lo tanto, si elegimos al tiempo como variable de etapa, esta característica nos impediría tratar problemas de tiempo mínimo y con ello no podríamos utilizar el atributo "general" en el enunciado de nuestro problema. Además de esta diferencia de adjetivo, es evidente que los problemas de tiempo mínimo son de importancia creciente en el mundo del control óptimo.

4.1.1.2 Elección de una Variable de Etapa Distinta del Tiempo

Ya que el tiempo no puede ser la variable de etapa, porque queremos que sea minimizado, nuestro objetivo es ahora determinar otra variable que sí pueda serlo. Por lo general las variables de estado de un sistema no son monótonas, sino que dependen de las otras y de los comandos. En caso de que existiera una variable con esas características, monotonía e independencia de las otras variables y del comando, el sistema tendría poco interés práctico. Parece entonces imposible conseguir otra variable de etapa.

La idea es que, aunque exista la dependencia, podamos determinar regiones del espacio de estados en que la monotonía de alguna de las variables esté garantizada. Del ejemplo de sistema (4.1), se ve que x_1 será monótona creciente allí donde x_2 sea positiva y será monótona decreciente allí donde x_2 sea negativa. Si dividimos el espacio de estado en esas dos regiones (x_2 positiva y x_2 negativa), en cada una de ellas podremos aplicar la programación dinámica utilizando como variable de etapa a x_1.

Una característica que vemos en (4.1) es que en la expresión de la derivada x_1 no aparece el comando. Es fácil darse cuenta de que esto es una necesidad ya que, como en el caso de x_2, la dependencia con el comando hace imprevisible la dirección de avance de la variable.

4.1.1.3 Formulación del Teorema

Una vez expuestas estas ideas, pasamos a formalizarlas:

Teorema 4-1. *Sea un sistema continuo cuya ecuación dinámica es*

$$\dot{\underline{x}} = \underline{f}(\underline{x},\underline{u}) \tag{4.3}$$

y sean Ω^k las regiones del espacio de estados en donde es posible aplicar la PD. Es posible encontrar un conjunto de regiones Ω^k si existe una variable de estado x_i tal que su derivada no dependa explícitamente del vector comando $\underline{u}$, es decir, si $f_i(\underline{x},\underline{u})=f_i(\underline{x})$.

Prueba: Definamos M^k a la variedad lineal formada por los puntos $\underline{x}\in X$ para los cuales $f_i(\underline{x},\underline{u})=f_i(\underline{x})=0$. Esta variedad divide X en regiones compactas de dos clases

$$\Omega^{k+} = \left\{\underline{x}: f_i(\underline{x},\underline{u}) = f_i(\underline{x}) > 0\right\} \tag{4.4}$$

$$\Omega^{k-} = \left\{\underline{x}: f_i(\underline{x},\underline{u}) = f_i(\underline{x}) < 0\right\} \tag{4.5}$$

Para cualquiera de estas regiones $\dot{x}_i>0$ ó $\dot{x}_i<0$, implicando que x_i se incrementa o decrementa monótonamente, por lo tanto es posible aplicar la PD en ella.♦

Al aplicar la PD dentro de cada una de esas regiones los puntos de la variedad lineal M^k, que pertenezcan a la frontera de la región, serán tomados como puntos de partida o de llegada, según lo sugiera la dinámica del sistema. En caso de ser considerados puntos de llegada, también denominados puntos de retorno, el valor de la función de mínimo coste será fijado a priori o vendrá determinado por la realización del primer barrido de la PD en otra región.

4.1.1.4 Proceso Para la Aplicación del Teorema

La forma de proceder para resolver un problema de índices generales es la siguiente:

1- Observar las ecuaciones dinámicas del sistema y seleccionar las variables de estado cuya derivada no esté afectada por ninguno de los comandos.

2- Elegir de esas variables la más adecuada y determinar la variedad lineal M^k, es decir, los puntos para los que la función $f_i(\underline{x})$ sea cero.

3- Estudiar cada una de las regiones en que ha quedado dividido el espacio de estados X. Para cada una de ellas determinar el signo de $f_i(\underline{x})$. Con ello se conoce si la variable de etapa va a crecer o decrecer y, por lo tanto, de que manera se van a desarrollar el primer y segundo barrido de la PD.

4- Estudiando las ecuaciones del sistema y las condiciones del problema (existencia de punto final), determinar si los puntos frontera de cada región serán puntos de partida o de retorno para dicha región.

5- Determinar los valores de la función de mínimo coste $I(\underline{x},k)$ para los puntos de retorno. Si son puntos de partida de otras regiones habrá que esperar a que se haga el primer barrido en dichas regiones. En caso contrario asignar valores adecuados: 0 para los puntos finales, un cierto valor si las trayectorias pueden pasar por ellos o ∞ si no interesa que las trayectorias pasen.

6- Finalmente, aplicamos el primer barrido de la PD comenzando por las regiones que no tengan dependencia de ninguna otra. De esta manera obtendremos los valores de la función de mínimo coste para los puntos de retorno en la frontera de las otras regiones.

Veamos a continuación este proceso en un ejemplo sencillo. Sea el sistema con las ecuaciones dinámicas siguientes

$$\begin{cases} \dot{x}_1 = -a_1 x_1 + a_2 x_2^2 \\ \dot{x}_2 = a_3 x_2 + a_4 x_1^2 x_2 + a_5 u \end{cases} \tag{4.6}$$

donde los parámetros $a_1,\ldots,a_5$ son positivos. Queremos hacer un control con índice general y con punto final el origen. Aplicando el proceso expuesto:

1- De las ecuaciones (4.6) vemos que es x_1 la única variable de estado cuya derivada no está afectada por el comando.

2- Será, por tanto, x_1 nuestra variable de etapa. Los puntos que pertenecen a la variedad lineal M se determinan haciendo $-a_1x_1+a_2x_2^2=0$, lo que da lugar a que

$$M = \left\{(x_1, x_2): x_1 = (a_2/a_1)x_2^2\right\} \tag{4.7}$$

que forma una parábola horizontal que pasa por el origen y con las ramas hacia la derecha.

3- Esta variedad M divide al espacio de estados en dos regiones

$$\Omega^1 = \left\{(x_1, x_2): x_1 < a_2/a_1\, x_2^2\right\} \tag{4.8}$$

$$\Omega^2 = \left\{(x_1, x_2): x_1 > a_2/a_1\, x_2^2\right\} \tag{4.9}$$

que corresponden al exterior e interior de la parábola respectivamente. En Ω^1 tenemos que $f_1>0$ y con ello $\dot{x}_1>0$, lo que significa que el sistema se desplazará de izquierda a derecha y el primer barrido de la PD se ha de realizar de derecha a izquierda. En Ω^2 ocurre lo contrario: $\dot{x}_1<0$, x_1 decrece, el sistema se desplaza de derecha a izquierda y el primer paso de la PD se ha de realizar de izquierda a derecha.

4- Como nuestro punto final es el origen, para la región Ω^1 los puntos de M son de retorno, ya que cualquier trayectoria que saliera de esos puntos, al evolucionar hacia la derecha, se alejaría del punto final. Para la región Ω^2 los puntos de M son puntos de partida, ya que las trayectorias que partan de ellos se dirigirán hacia la izquierda acercándose hacia el origen, nuestro punto final.

5- Por lo dicho en el punto anterior, inicialmente sólo el origen tendrá determinado el valor de la función de mínimo costo, que tomaremos por conveniencia igual a cero. Para los puntos de M este valor se determinará al realizar el primer barrido de la PD en Ω^2.

6- Comenzaremos a aplicar la PD en la región Ω^2 porque tenemos determinado el valor del costo mínimo para sus puntos finales, en este caso únicamente el origen. El primer barrido lo realizaremos de izquierda a derecha y, al considerar el resto de los puntos de M como puntos de salida, obtendremos para ellos el valor de la función de mínimo costo. Una vez determinados estos valores podemos aplicar la PD a la región Ω^1 ya que tenemos los valores de $I(\underline{x},k)$ para sus puntos finales.

Al realizar el segundo barrido, es decir, al reconstruir la trayectoria óptima, pasaremos de una región a otra a través de los puntos de M. Si la trayectoria comienza en Ω^1

alcanzaremos algún punto de *M*. Si este punto es el origen tendremos la trayectoria completa. En caso contrario, continuaremos la trayectoria en Ω^2 teniendo como punto de partida dicho punto de *M*. En este caso llegaremos al origen ya que es el único punto final para esta región. En la figura 4.1 se presenta un esquema de los elementos determinados para este sistema.

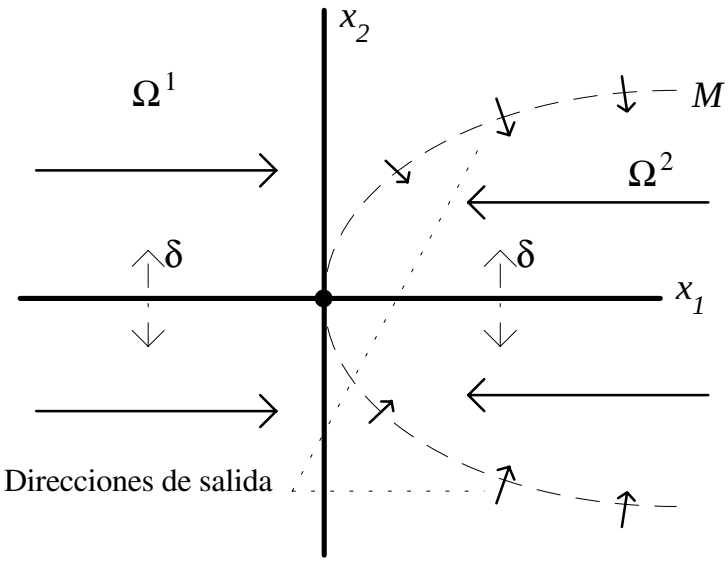

Figura 4.1 - Regiones y Simetría del Ejemplo

4.1.2 Simetría de los Sistemas

Una de las formas de reducir la complejidad de la PD es determinar un tipo de simetría en el sistema que nos permita extender los resultados obtenidos en una zona del espacio de estado a otras. Suponiendo que la dinámica del sistema para un estado es igual a la de otro estado tras una transformación, si conocemos el comando óptimo para uno de dichos estados aplicando esa transformación tendremos el comando óptimo para el otro. La justificación de esta afirmación la haremos basándonos en la invarianza del Hamiltoniano. Enunciaremos esta idea mediante el siguiente teorema:

Teorema 4-2. *Sea un problema de control óptimo para un sistema cuya ecuación dinámica es*

$$\dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u}) \tag{4.10}$$

donde los estados han de pertenecer al conjunto X y los comandos al conjunto U; y sea la función de costo de la forma

$$J = \int_{t_i}^{t_f} L(\underline{x}, \underline{u}) dt \tag{4.11}$$

Dada una región $A \subset X \times U$, donde hemos calculado la PD, definiremos la región $\overline{A} \subset X \times U$, complemento de A, si $\forall (\underline{x},\underline{u}) \in A$, $\exists$ una transformación δ tal que

$$\begin{aligned} \delta &: A \to \overline{A} \\ (\underline{x},\underline{u}) &\to (\hat{\underline{x}},\hat{\underline{u}}) = \delta(\underline{x},\underline{u}) \end{aligned} \tag{4.12}$$

cumpla que $f(\underline{x},\underline{u})=f(\hat{\underline{x}},\hat{\underline{u}})$ y $L(\underline{x},\underline{u})=L(\hat{\underline{x}},\hat{\underline{u}})$. En las regiones A y $\overline{A}$ la PD da la misma información para cada punto. Consecuentemente es suficiente aplicar la PD en A.

Prueba: Del principio de mínimo de Pontryagin sabemos que el control óptimo es aquel que minimiza el Hamiltoniano. Éste tiene la forma

$$H(\underline{x},\underline{u},\underline{\lambda}) = L(\underline{x},\underline{u}) + \underline{\lambda}^{\mathrm{T}}(t)\,\underline{f}(\underline{x},\underline{u}) \tag{4.13}$$

donde a su vez $\underline{\lambda}$ se obtiene como

$$\dot{\underline{\lambda}} = -\frac{\partial H(\underline{x},\underline{u},\underline{\lambda},t)}{\partial \underline{x}} \tag{4.14}$$

La transformación δ por hipótesis mantiene invariante a $\underline{f}$ y L, y de las expresiones anteriores se deduce que δ mantendrá por tanto invariante al Hamiltoniano. Por ello, se concluye que los comandos óptimos para puntos en $\overline{A}$ se pueden obtener aplicando la transformación δ a los comandos óptimos de los puntos complementarios en *A*. Por ello la información que nos da la PD en $\overline{A}$ la podemos obtener a partir de la que nos da en *A*.♦

De esta forma, si encontramos la transformación δ sólo tendremos que aplicar el primer barrido de la PD a una de las regiones, y además la información necesaria para la otra región se obtiene fácilmente de estos resultados. Habremos reducido en un factor 2 la complejidad tanto temporal como espacial.

Hacer notar que en numerosísimos casos las funciones de costo son invariantes ante las transformaciones empleadas en este teorema. Por ejemplo, índices cuadráticos o de valor absoluto serán invariantes a cambios de signo. Un ejemplo claro de invarianza es el caso de los índices de tiempo mínimo. Como en ese caso se pesa el tiempo total empleado en la trayectoria este valor no es afectado por ningún cambio en las variables de estado ni en los comandos. En estos casos de invarianza del costo bastará centrar la atención en la invarianza de la ecuación dinámica del sistema.

4.1.2.1 Proceso para la Aplicación del Teorema

El proceso completo para aplicar esta reducción será el siguiente:

1- Observar las ecuaciones dinámicas del sistema y la función de costo para determinar si existe la transformación δ que mantenga invariante dichas ecuaciones.

2- Dividir el espacio de trabajo X en dos regiones que sean complementarias mediante la transformación δ.

3- Calcular y almacenar los resultados del primer paso de la PD únicamente para una de esas regiones.

4- Durante el segundo paso la PD, si nos encontramos en un punto de la región complementaria, para la cual no disponemos de información, calcular el punto complementario, del cual sí tendremos información, y aplicar el comando óptimo complementario al de éste.

Para el sistema ejemplo (4.6) este procedimiento será entonces:

1- Observando las ecuaciones (4.6) vemos que si hacemos la transformación

$$(x_1, x_2, u) \rightarrow (x_1, -x_2, -u) \tag{4.15}$$

dichas ecuaciones no cambian, ya que los cambios de signo se anulan. Suponiendo que esta transformación mantiene invariante también la función de costo, ésta será nuestra transformación δ.

2- Para cualquier punto su complementario será el simétrico con respecto al eje x_2. Tomaremos al semiplano superior como una de nuestras regiones, y al semiplano inferior como la complementaria.

3- Realizamos ahora el primer paso de la PD en alguno de esos semiplanos, por ejemplo el superior, y almacenamos los resultados.

4- Si al reconstruir la trayectoria óptima nos encontramos en (x_1,x_2) con $x_2<0$, es decir perteneciente al semiplano inferior, con δ obtenemos que su complementario es el punto $(x_1,-x_2)$. Como tenemos almacenado $u^o(x_1,-x_2)$, por ser un punto del semiplano superior, el comando a aplicar será éste cambiado de signo, según indica δ.

En la figura 4.1 se muestra también la forma en que se aplica la simetría en este ejemplo.

Este método es totalmente compatible con el presentado en la sección anterior para la elección de la variable de etapa. En algunos sistemas las distintas regiones Ω son a la vez regiones complementarias. Por lo general la simetría dividirá una única región Ω en dos subregiones complementarias, con lo cual la PD se realizará en el orden necesario pero sólo en una de las subregiones de cada Ω.

4.1.3 Eliminación de la Interpolación

Como vimos en la sección 1.3.4, en el proceso del primer barrido de la PD, por lo general, será necesario interpolar. Como se indicó, esta necesidad surge porque, al aplicar uno de los comandos discretizados a nuestro estado actual, el estado al que se llega en la etapa siguiente puede no ser uno de los considerados en la discretización.

Si la ecuación dinámica discretizada del sistema se pudiese modificar de manera que, dado el estado actual y el siguiente, ésta nos diera el comando necesario para realizar dicha transición, el problema de la interpolación se podría evitar.

4.1.3.1 Modificación del Primer Barrido

Supongamos que disponemos de la función $\underline{f}^{-1}$ tal que $\underline{u}_k$ se puede obtener como

$$\underline{u}_k = \underline{f}^{-1}(\underline{x}_k, \underline{x}_{k+1}, \underline{u}_{k+1}, k) \tag{4.16}$$

donde, en el caso más general, puede depender del comando que se dará en la etapa siguiente $\underline{u}_{k+1}$. Éste estará disponible al realizar la programación dinámica hacia atrás. Basándonos en la existencia de esta función, el proceso del primer barrido de la PD que evita la interpolación se describe a continuación.

Como en el caso general, debemos discretizar el conjunto de variables de estado permitidas para cada etapa *X(k)* en *P(k)* posibles valores

$$X(k) = \left\{\underline{x}^1, \underline{x}^2, \ldots, \underline{x}^{P(k)}\right\} \tag{4.17}$$

En cambio no es necesario discretizar el conjunto de comandos posibles *U(x,k)*. Para la etapa *k*, suponiendo que tenemos todos los *I(x,k+1)*, para cada estado $\underline{x}$ de esta etapa se irán eligiendo los estados discretizados de la etapa siguiente $\underline{x}^p \in X(k+1)$. Para cada uno de estos estados calcularemos el comando necesario para realizar la transición del estado actual al estado elegido mediante

$$\underline{u}^p = \underline{f}^{-1}(\underline{x}_k, \underline{x}^p, \underline{u}_{k+1}, k) \tag{4.18}$$

donde $\underline{u}_{k+1}$ es el comando óptimo conocido para $\underline{x}^p$. Si este comando no es permitido, es decir, no pertenece al conjunto *U(x,k)*, descartaremos al estado $\underline{x}^p$ como posible estado siguiente óptimo. En el caso contrario, podremos calcular el costo total de elegir el estado siguiente $\underline{x}^p$ como

$$F^p = L(\underline{x}, \underline{u}^p, k) + I(\underline{x}^p, k+1) \tag{4.19}$$

Esto lo podemos hacer sin necesidad de interpolar ya que *I(x^p,k+1)* está siempre disponible por ser $\underline{x}^p \in X(k+1)$. Una vez tengamos esta magnitud para los *P(k+1)* estados siguientes podremos pasar a minimizar por comparación. Si el valor mínimo se da para F^q, será $I(\underline{x},k)=F^q$ y $\underline{u}^o(\underline{x},k)=\underline{u}^q$.

Una vez realizado esto para todos los estados de la etapa *k* tendremos disponibles todos los $I(\underline{x},k)$ y podremos proceder para la etapa *k-1* y así, sucesivamente, hasta que lleguemos a los puntos de partida.

En el caso de un índice general, para el cual debamos tomar una variable de etapa distinta del tiempo, es necesario que la ecuación de estados inversa nos dé, no sólo el comando, sino también el tiempo para hacer la transición de un estado a otro, es decir,

$$\underline{v} = (\underline{u}_k, T_k) = \underline{f}^{-1}(\underline{x}_k, \underline{x}_{k+1}, \underline{u}_{k+1}, k) \tag{4.20}$$

En este caso el intervalo de tiempo es totalmente equivalente a un comando más. Por lo tanto, lo que dijimos para el comando en el proceso anterior es extensible al par $(\underline{u}_k, T_k)$. Una restricción clara a imponer al intervalo de tiempo es que sea siempre positivo.

4.1.3.2 La Discretización BPF

Como vimos en el apartado 3.2.2, al presentar distintas discretizaciones, la BPF daba lugar a expresiones más completas. Éstas tenían en cuenta ambos extremos del intervalo. Por esta propiedad es una discretización muy adecuada para obtener la función $\underline{f}^{-1}$.

Si aplicamos la BPF al sistema (4.6) tendremos las expresiones

$$\begin{cases} 2(x_{1k+1} - x_{1k})/T_k = -2a_1 + a_2(x_{2k}^2 + x_{2k+1}^2) \\ 2(x_{2k+1} - x_{2k})/T_k = a_3(x_{2k} + x_{2k+1}) + a_4(x_{1k}^2 x_{2k} + x_{1k+1}^2 x_{2k+1}) + a_5(u_k + u_{k+1}) \end{cases} \tag{4.21}$$

donde hemos hecho la aproximación $T_{k+1} \approx T_k$. Podemos despejar de la primera una expresión para T_k

$$T_k = \frac{2(x_{1k+1} - x_{1k})}{-2a_1 + a_2(x_{2k}^2 + x_{2k+1}^2)} \tag{4.22}$$

y de la segunda para u_k

$$u_k = \left[2(x_{2k+1} - x_{2k})/T_k - a_3(x_{2k} + x_{2k+1}) - a_4(x_{1k}^2 x_{2k} + x_{1k+1}^2 x_{2k+1})\right] / a_5 - u_{k+1} \tag{4.23}$$

Estas dos ecuaciones constituyen las componentes de $\underline{f}^{-1}$ para este sistema y nos posibilitan la aplicación del procedimiento sin discretización.

Además, en los trabajos previos [Acos 91] [More 92] se demostró que este tipo de discretización permitía reducir los puntos de la discretización, es decir, aumentar el intervalo de separación entre ellos, sin pérdida de la optimalidad. De esta manera es posible reducir la complejidad espacial porque con menos puntos se cubre una misma región del espacio de estado. A la vista de esta propiedad intentaremos utilizar, allí donde sea posible, esta discretización preferiblemente a cualquier otra.

4.2 Aplicación a los Problemas Ejemplo

Veremos ahora la aplicación conjunta de todas las técnicas presentadas en la sección anterior a nuestros sistemas ejemplo. Como los problemas están planteados para tiempo mínimo la función de costo será invariante ante cualquier transformación y sólo debemos considerar las ecuaciones dinámicas de los sistemas para aplicar el teorema **4-2**.

4.2.1 Primer Sistema

Recordemos las expresiones para nuestro primer sistema, planteadas en la sección 3.3.1. Éstas son

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = ax_2 + bx_2x_1^2 + dx_1 + cu \end{cases} \tag{3.28}$$

Para este sistema planteamos un problema con estado final el origen y una función de coste de tiempo mínimo.

4.2.1.1 Elección de la Variable de Etapa

Como queremos resolver problemas de tiempo mínimo debemos elegir una variable de etapa distinta del tiempo. Hacemos uso del teorema **4-1** para aplicar la PD.

Observando las ecuaciones (3.28) vemos que para la variable x_1 su derivada no está afectada por el comando. Cumple por ello las condiciones del teorema y será nuestra variable de etapa.

Su derivada coincide con x_2 por lo que se hará cero cuando ésta se haga cero. La variedad M es entonces el eje x_1. Esto divide al plano fásico en dos semiplanos. Nuestras regiones de trabajo serán

$$\Omega^1 = \{(x_1, x_2) : x_2 < 0\} \tag{4.24}$$

$$\Omega^2 = \{(x_1, x_2) : x_2 > 0\} \tag{4.25}$$

Para el semiplano inferior (Ω^1), con valores de x_2 negativos, x_1 será decreciente y el sistema evoluciona de derecha a izquierda. Para el semiplano superior (Ω^2), con valores de x_2 positivos, x_1 será creciente por lo que el sistema evoluciona de izquierda a derecha.

Como deseamos como punto final el origen, éste tendrá un valor para la función de mínimo coste igual a cero. En cuanto a la región Ω^1, los puntos de la frontera que están a la derecha del origen, es decir, el semieje x_1 positivo, serán puntos de salida ya que la evolución en esa región, de derecha a izquierda, los acercará hacia el punto final. Los puntos del semieje x_1 negativo serán puntos de llegada ya que, por las mismas

circunstancias, las trayectorias que salieran de ellos se alejarían del punto final. Aplicando el mismo razonamientos para Ω^2 vemos que ocurre lo contrario: los puntos del semieje x_1 positivo han de ser de llegada mientras que los del semieje x_1 negativo han de ser de salida.

Como acabamos de decir, ocurre que los puntos de llegada de una región son los de salida de la otra. Existe una dependencia mutua entre ambas, es decir, ninguna de las dos tiene todos sus puntos finales determinados. Para romper esa dependencia, y poder comenzar a aplicar el primer barrido de la PD, fijamos un valor de $I(\underline{x},k)$ para los puntos de retorno. Este valor puede ser arbitrario, incluso cero, sin que afecte al resultado final. Una vez fijado este valor podemos comenzar por cualquiera de las dos regiones.

4.2.1.2 Propiedad de Simetría

Observando las ecuaciones (3.28) vemos que si cambiamos, a la vez, el signo de todas las variables se anulan los cambios de signo dando lugar a las mismas ecuaciones. Por ello nuestra transformación δ se definirá como

$$\delta(x_1, x_2, u) = (-x_1, -x_2, -u) \tag{4.26}$$

Esta transformación representa una simetría respecto al origen. Las regiones complementarias serán cualquier pareja de semiplanos cuya frontera pase por el origen. Por conveniencia, ya que nuestras regiones Ω^1 y Ω^2 cumplen esa condición, las elegimos como regiones complementarias, es decir, Ω^1 será complementaria de Ω^2.

Realizando el primer paso de la PD en Ω^1 tendremos los resultados para Ω^2. Ésta fue la alternativa que elegimos al realizar los cálculos. Si queremos obtener la trayectoria para un punto del semiplano superior bastará con obtenerla para su punto simétrico

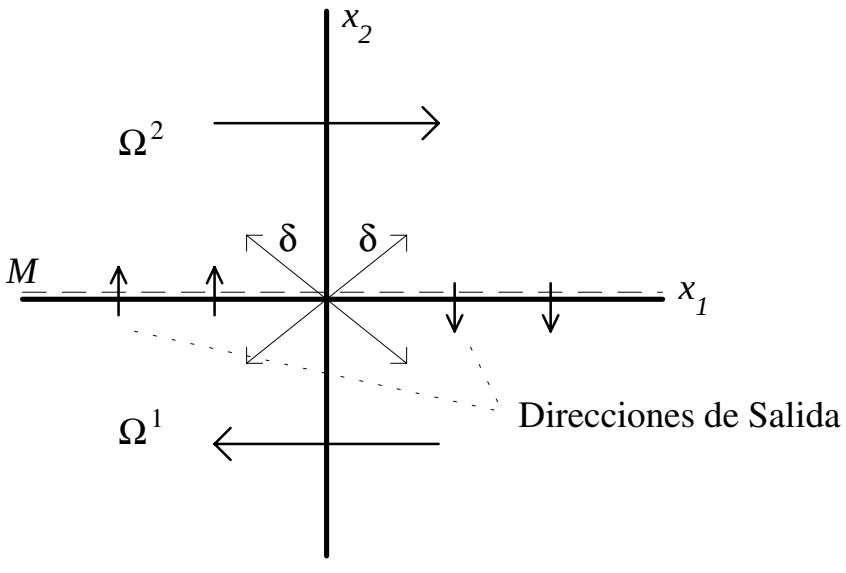

Figura 4.2 - Regiones y Simetría del Primer Sistema

respecto al origen, el cual estará en el semiplano inferior, y luego transformar mediante δ todos los estados y comandos aplicados.

El esquema de las regiones y simetría del sistema se muestra en la figura 4.2.

4.2.1.3 Expresiones para el Comando y el Tiempo

Si queremos aplicar el método de la PD que elimina la necesidad de interpolar en el primer barrido debemos llegar a expresiones discretizadas en que se obtengan el comando y el intervalo de tiempo en función del estado inicial y final.

Se pueden obtener las expresiones necesarias aplicando la formulación BPF que, por las propiedades mencionadas en el apartado 4.1.3.2, las preferimos a las que se pueden obtener mediante discretización clásica. Las ecuaciones de este sistema discretizadas mediante BPF eran:

$$\begin{cases} 2(x_{1k+1}-x_{1k}) = T_k(x_{2k}+x_{2k+1}) \\ 2(x_{2k+1}-x_{2k}) = T_k\left[a(x_{2k}+x_{2k+1}) + b(x_{2k}x_{1k}^2 + x_{2k+1}x_{1k+1}^2) + d(x_{1k}+x_{1k+1}) + c(u_k+u_{k+1})\right] \end{cases} \quad (3.35)$$

De la primera podemos despejar T_k

$$T_k = \frac{2(x_{1k+1}-x_{1k})}{(x_{2k}+x_{2k+1})} \quad (4.27)$$

y de la segunda u_k

$$u_k = \frac{1}{c}\left[\frac{2(x_{2k+1}-x_{2k})}{T_k} - a(x_{2k}+x_{2k+1}) - b(x_{2k}x_{1k}^2 + x_{2k+1}x_{1k+1}^2) - d(x_{1k}+x_{1k+1})\right] - u_{k+1} \quad (4.28)$$

expresión que, por sencillez, dejamos en función de T_k. Como valor para u_{k+1} utilizaremos el del comando óptimo que se ha calculado para el estado $\underline{x}_{k+1}$ que estamos probando.

4.2.1.4 Resultados Obtenidos

Realizamos un estudio de este sistema para la región Ω^1. Para la variable de etapa (x_1) tomamos una discretización uniforme de incremento $d_1=0.02$ con 100 puntos en el semieje positivo y 100 puntos en el semieje negativo. Con esto cubrimos el intervalo [*-2,2*]. Para la otra variable de estado tomamos la misma discretización ($d_2=0.02$) y 100 puntos en el semieje negativo, cubriendo el intervalo [*-2,0*].

El número total de puntos de que consta la discretización utilizada es de 101× 201=20.301. Esto hace necesario un espacio de almacenamiento de aproximadamente 317Kb, ya que guardamos dos valores en doble precisión (u_k y T_k) que ocupan 8 bytes

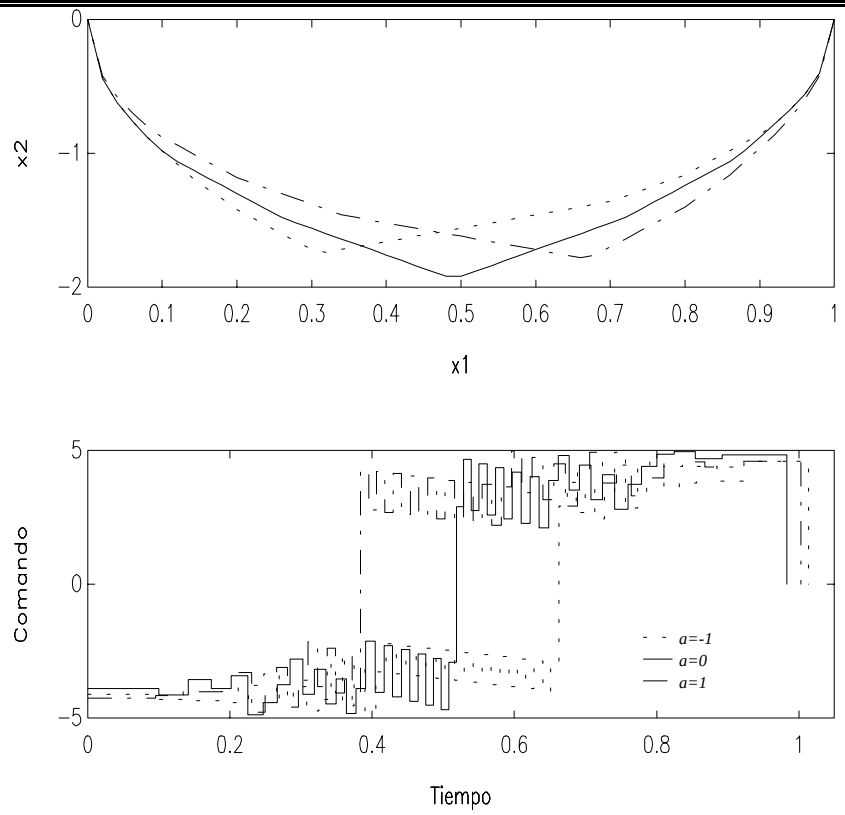

Figura 4.3 - Trayectorias al variar el parámetro *a*

cada uno. Durante el cálculo, al tener que almacenarse también *I*, los requerimientos de memoria ascienden a 476Kb. El tiempo de cálculo es proporcional a esta cantidad.

Sabemos, del caso lineal, que las trayectorias de tiempo mínimo que parten de un punto sobre el eje x_1 presentan un tramo de alejamiento del eje x_1 en el que se aplica el máximo comando. Cuando se alcanza la denominada curva de conmutación, se regresa, con el máximo comando de signo opuesto, al origen.

En la figura 4.3 se muestran las trayectorias para distintos valores del parámetro *a*, desde el valor para el sistema lineal *a=1* hasta el del sistema *Van der Pol a=-1*. Se observa como se modifica la forma de la curva de conmutación.

En la figura 4.4 se muestra la evolución de las trayectorias para distintos valores del parámetro *b*, desde el sistema lineal *b=0*, hasta un sistema en que se pesa fuertemente el término no lineal *b=10*. Con el crecimiento de ese término el sistema se aleja menos del eje y por ello tarda más en llegar al origen.

La evolución de las trayectorias al ir aumentando el valor del parámetro *d* se muestra en la figura 4.5. Al igual que ocurría con el parámetro *b*, el sistema no está tan lejos del eje cuando encuentra la curva de conmutación, por lo que tarda más tiempo en alcanzar el origen.

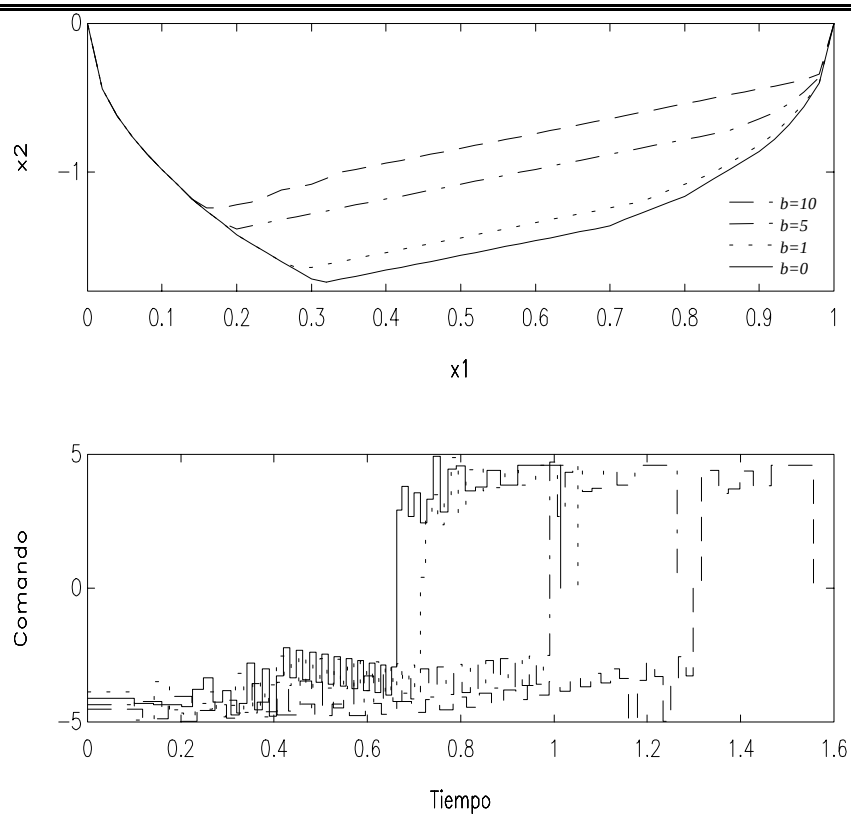

Figura 4.4 - Trayectorias al variar el parámetro *b*

Por lo general la adición de los nuevos términos al sistema lineal lo hacen más lento, ya que en el tramo de bajada se oponen a la acción del comando negativo. En el tramo de subida esta acción se suma al comando positivo empleado pero, al ser ya x_1 pequeño, su importancia es menor. A pesar de estos factores el sistema sigue presentando la curva de conmutación, típica del sistema lineal aunque situada en diferente posición.

4.2.2 Segundo Sistema

Como vimos en la sección 3.3.2 nuestro segundo sistema es de tercer orden con dos comandos. Optamos por el sistema alternativo de ecuación

$$\begin{cases} \dot{x}_1 = \cos(x_3)u_1 \\ \dot{x}_2 = a + \operatorname{sen}(x_3)u_1 \\ \dot{x}_3 = u_2 \end{cases} \tag{3.37}$$

Queremos obtener la trayectoria de tiempo mínimo con punto final el origen del plano (x_1,x_2). Pasaremos a estudiar, a continuación, la aplicación de la PD a este sistema.

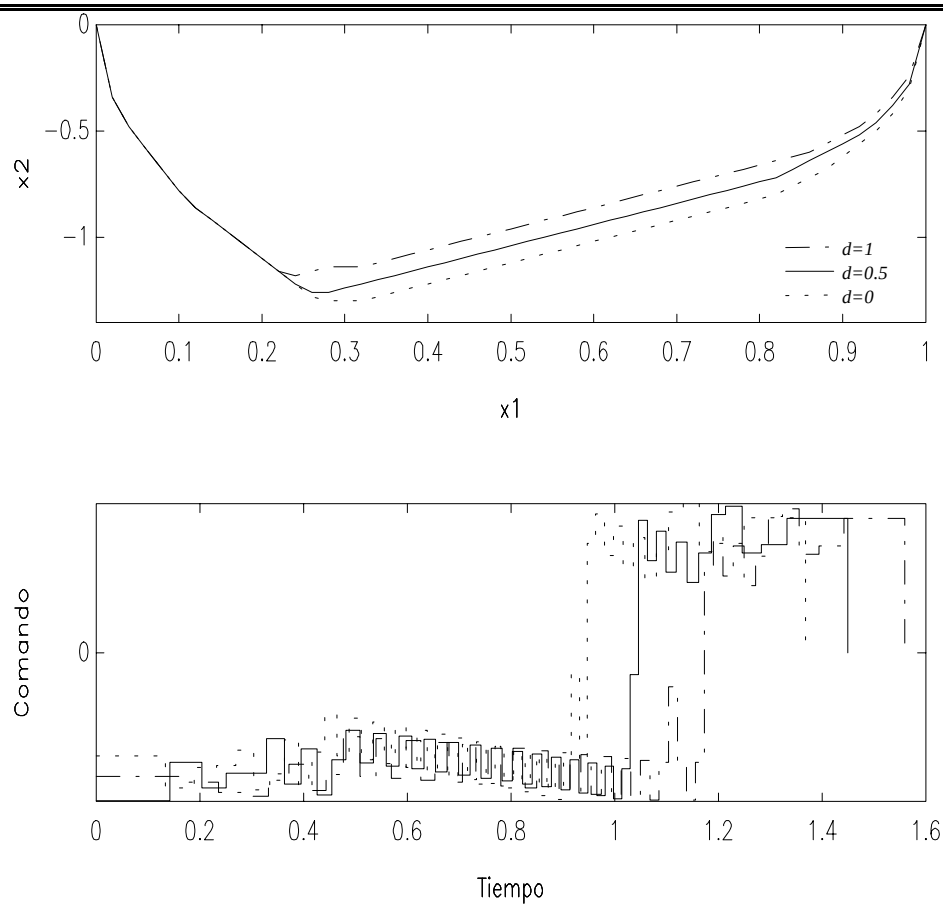

Figura 4.5 - Trayectorias al variar el parámetro *d*

4.2.2.1 Elección de la Variable de Etapa

Estudiando las ecuaciones (3.37) observamos que no existe ninguna variable para la cual su derivada no esté afectada por ningún comando. No tenemos la posibilidad de aplicar el teorema **4-1**. A pesar de ello, podemos aplicar el mismo razonamiento que utilizamos al deducir aquel.

Lo que necesitamos es una variable que sea monótona al menos en determinadas regiones. Para ello, la expresión a que se iguala su derivada con respecto al tiempo debe tener signo independiente del comando, para que sea monótona y se pueda definir el concepto de etapa. Habíamos indicado que el robot sólo podía avanzar, con lo cual el comando u_1 es siempre positivo. Por esto, el signo de las dos primeras ecuaciones dependerá únicamente de la variable x_3. Tanto x_1 como x_2 podrán ser nuestras variables de etapa.

Por llegarse a regiones más sencillas, elegimos x_1 para tal fin. La variedad M será la formada por los puntos en los que f_1 sea 0. Al estar x_3 como argumento de un coseno en f_1 éste se hará 0 cuando $x_3=\pi/2+k\pi$, con $k\in Z$. La variedad M estará formada por los planos en los cuales x_3 cumpla esta condición, es decir

$$M = \bigcup_{k \in Z} M^k = \bigcup_{k \in Z} \{(x_1, x_2, x_3): x_3 = -\pi/2 + k\pi\} \tag{4.29}$$

El espacio de estados quedará dividido en dos tipos de regiones

$$\Omega_k^+ = \{(x_1, x_2, x_3): -\pi/2 + 2k\pi < x_3 < \pi/2 + 2k\pi\} \tag{4.30}$$

$$\Omega_k^- = \{(x_1, x_2, x_3): -3\pi/2 + 2k\pi < x_3 < -\pi/2 + 2k\pi\} \tag{4.31}$$

En las regiones Ω^+, f_1 será positivo y por lo tanto x_1 crecerá, es decir, las trayectorias irán de valores de x_1 menores a valores de x_1 mayores. El primer barrido de la PD habrá de hacerse en sentido contrario. En las regiones Ω^- ocurre lo inverso y por lo tanto x_1 decrecerá.

El conjunto de puntos finales corresponde al eje x_3, ya que no importa la dirección en que quede el robot al llegar al origen. Para las regiones Ω^+ estos estados no serán finales ya que el sistema evoluciona aumentando x_1 y no reduciéndolo y por lo tanto se aleja del eje x_3. Los puntos finales, entonces, se encuentran únicamente en las regiones Ω^-. Los puntos de las fronteras M^k serán puntos de salida con respecto a las regiones Ω^-, mientras que representarán los puntos de retorno para las regiones Ω^+. En la figura 4.6 se intenta esquematizar esta situación.

Para comenzar la realización de la PD debemos hacerlo por cualquiera de las regiones Ω^- ya que son completamente independientes. Para poder realizar la PD en una región Ω^+ es necesario que se haya realizado ya en las regiones adyacentes, para que los puntos de la frontera tengan determinada su función de mínimo costo.

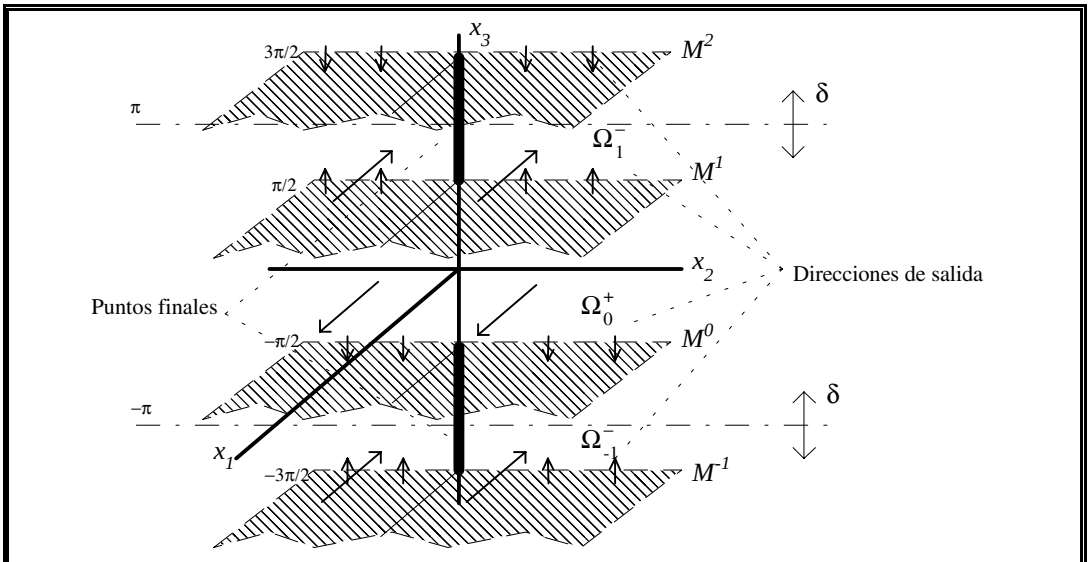

Figura 4.6 - Esquema de las Regiones del Segundo Sistema

4.2.2.2 Propiedad de Simetría

Por ser las funciones seno y coseno de periodo 2π, si hacemos la transformación

$$(x_1, x_2, x_3, u_1, u_2) \rightarrow (x_1, x_2, x_3 + 2k\pi, u_1, u_2) \tag{4.32}$$

donde $k \in Z$, las ecuaciones del sistema no se modifican. Esto significa que los resultados calculados para la banda $x_3=[-\pi,\pi]$ serán extensibles al resto del espacio de estado. Realizaremos los cálculos en esta banda. De cualquier manera, por la periodicidad de las funciones trigonométricas, cualquier punto inicial con un valor de x_3 fuera de este rango será equivalente a algún punto de esta banda.

Las distintas regiones y direcciones deducidas para este sistema se esquematizan en la figura 4.6.

4.2.2.3 Expresiones para el Comando y el Tiempo

Podemos obtener las expresiones necesarias de la discretización BPF. Las ecuaciones discretizadas para este sistema son

$$\begin{cases} 2(x_{1k+1} - x_{1k})/T_k = \cos(x_{3k})u_{1k} + \cos(x_{3k+1})u_{1k+1} \\ 2(x_{2k+1} - x_{2k})/T_k = a + \operatorname{sen}(x_{3k})u_{1k} + \operatorname{sen}(x_{3k+1})u_{1k+1} \\ 2(x_{3k+1} - x_{3k})/T_k = u_{2k} + u_{2k+1} \end{cases} \tag{4.33}$$

Entre las dos primeras podemos encontrar una expresión para el intervalo de tiempo y el comando u_1 en la etapa k

$$T_k = 2\frac{(x_{2k+1} - x_{2k}) - \tan(x_{3k})(x_{1k+1} - x_{1k})}{2a + \operatorname{sen}(x_{3k+1})u_{1k+1} - \tan(x_{3k})\cos(x_{3k+1})u_{1k+1}} \tag{4.34}$$

$$u_{1k} = \frac{1}{\cos(x_{3k})}\left[2\frac{x_{1k+1} - x_{1k}}{T_k} - \cos(x_{3k+1})u_{1k+1}\right] \tag{4.35}$$

y de la tercera la expresión de u_2

$$u_{2k} = 2\frac{x_{3k+1} - x_{3k}}{T_k} - u_{2k+1} \tag{4.36}$$

4.2.2.4 Resultados Obtenidos

Para estudiar este sistema elegimos una discretización uniforme en todas las variables. El rango de x_3 que debemos cubrir es $[-\pi,\pi]$, en el que colocamos 121 puntos, dando lugar a una discretización $d_3=0.052$. Como x_1 lo considerábamos sólo positivo, como si existiera una pared, tomamos 110 puntos en el semieje positivo separados $d_1=0.005$ cubriendo con ello el rango $[0,0.55]$. Para x_2 tomamos 5 puntos en la dirección negativa y 80 en la positiva con una separación de $d_2=0.045$ cubriendo el rango $[-0.225,3.6]$. Esta

asimetría en x_2 es debido a que el punto inicial lo tomaremos sobre el semieje positivo y las trayectorias óptimas no llegarán a grandes valores negativos de x_2.

Con este número de puntos tenemos una rejilla de 121×110×85=1.131.350 puntos. Como debemos almacenar 3 valores de doble precisión (u_{1k}, u_{2k} y T_k) por punto, esto significa una necesidad de memoria de aproximadamente 26Mb. Durante el proceso de cálculo, además, debemos almacenar el costo mínimo I, haciendo entonces un total superior a los 34Mb.

Elegimos para el parámetro a un valor de *0.01* y unas cotas máximas para el avance de *10* y para el giro de *5*, con lo cual $U=\{[0,10]\times[-5,5]\}$. No es necesario poner cotas al tiempo ya que, al ser el objeto de la minimización, tendrá valores pequeños.

En la figura 4.8 se muestran las trayectorias para distintos puntos iniciales sobre el eje x_2. Vemos que para el avance no se utilizan los comandos máximos, como cabría esperar en una trayectoria de tiempo mínimo. Este fenómeno se lo achacamos a una deficiencia en la discretización ya que sólo se usan tres valores en la dirección x_1. Sería necesario reducir el paso de discretización tanto en la dirección x_1 como x_2, pero, por problemas de memoria, nos resulta imposible. Por la misma causa, ocurre que trayectorias desde puntos más lejanos al punto final invierten menos tiempo en llegar al origen (figura 4.9). El motivo es que, en las trayectorias que parten de puntos más alejados, se utiliza más avance que en las que parten del eje x_2.

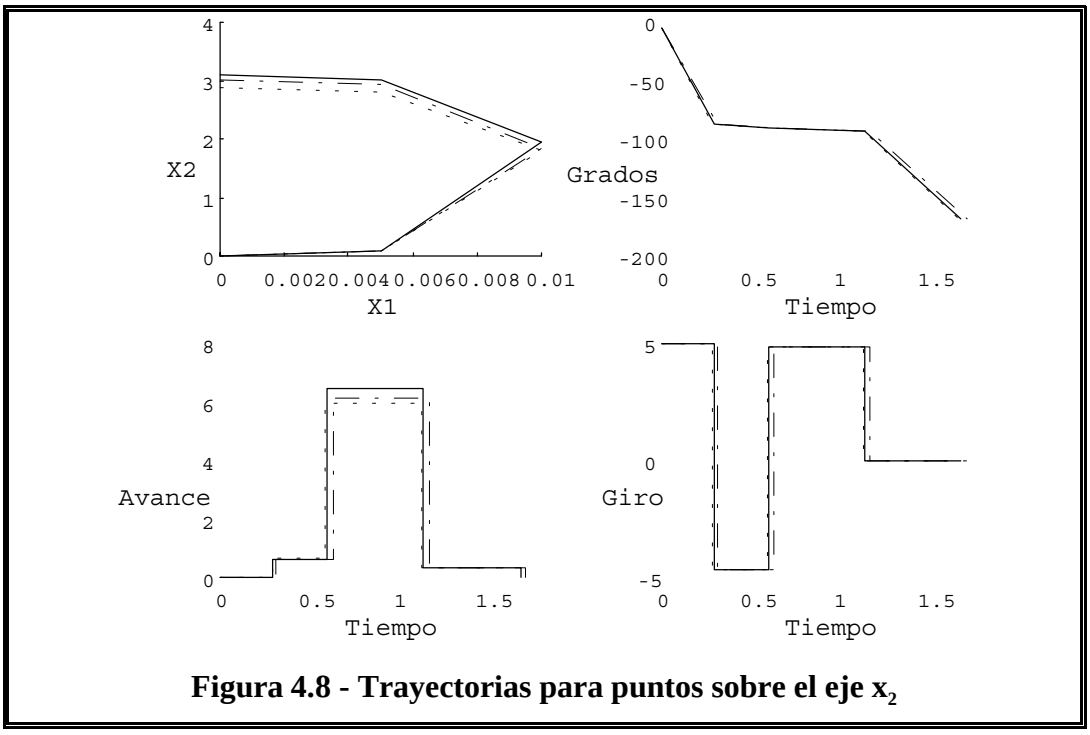

Figura 4.8 - Trayectorias para puntos sobre el eje x_2

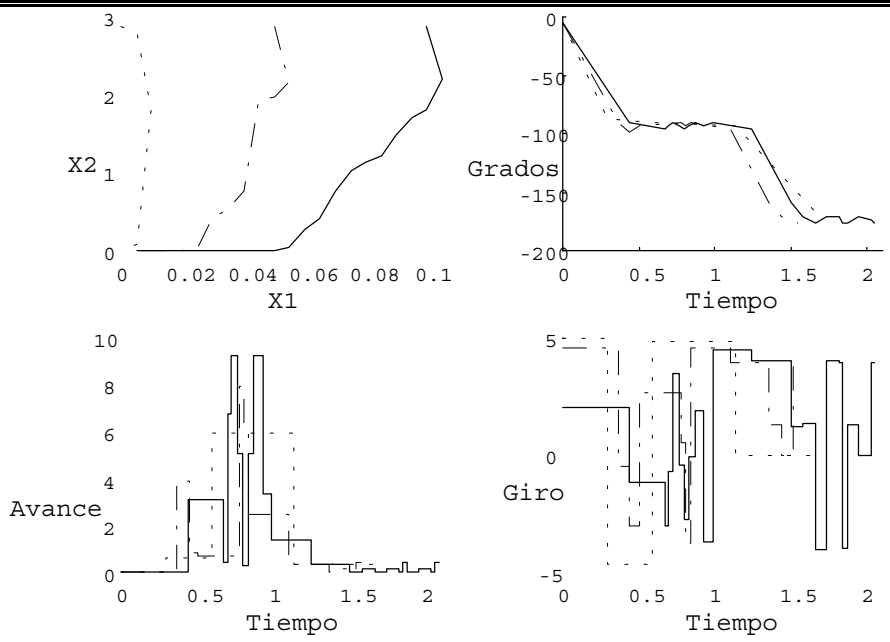

Figura 4.9 - Comparación con trayectorias para puntos fuera del eje x_2

4.2.3 Tercer Sistema

Para el tercer sistema, en la sección 3.3.3, habíamos obtenido el modelo matemático siguiente

$$\begin{cases} \dot{h}_1 = -R_1 P(h_1 - h_2 + h_0) + q_l \\ \dot{h}_2 = R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2) \\ \dot{\theta} = \left\{ \theta \left[-1/R_T - C_p \left(R_1 P(h_1 - h_2 + h_0) - R_2 P(h_2) \right) \right] + q_c \right\} \Big/ C_s \end{cases} \tag{3.66}$$

A este sistema queremos aplicarle también índices de tiempo mínimo y conseguir que llegue a un estado tal que el segundo depósito esté suministrando un determinado caudal a una determinada temperatura. Por la relación que existe entre caudal y altura de los depósitos, esto es equivalente a fijar los valores de h_2 y θ. Por la relación que existe entre h_1 y h_2, si queremos que h_2 quede estable, h_1 ha de quedar también en un valor determinado. En definitiva lo que tenemos es un único estado final.

4.2.3.1 Elección de la Variable de Etapa

De la ecuación (3.66) observamos que h_2 es la única variable cuya derivada no está afectada por ningún comando. Por el teorema **4-1** podrá ser nuestra variable de etapa.

Los puntos para los cuales f_2 se hace *0* formarán la variedad *M*. En este caso

$$M = \{(h_1, h_2, \theta) : h_1 = (R_2/R_1)h_2 - h_0\} \tag{4.37}$$

Esto representa un plano paralelo a la dirección del eje θ. Como al punto final le exigimos que sea estable, es decir, que no siga evolucionando, esto implicará que h_2 no ha de variar con el tiempo y por lo tanto el punto final estará contenido en este plano.

El plano *M* divide nuestro espacio de trabajo en dos regiones

$$\Omega^1 = \{(h_1, h_2, \theta): h_1 < (R_2/R_1)h_2 - h_0\} \tag{4.38}$$

$$\Omega^2 = \{(h_1, h_2, \theta): h_1 > (R_2/R_1)h_2 - h_0\} \tag{4.39}$$

En Ω^1 la variable h_2 es decreciente por ser $f_2<0$. En Ω^2, en cambio, h_2 es creciente. Como vamos a aplicar la PD hacia atrás, el primer barrido se ha de realizar en el sentido contrario al avance de nuestra variable de etapa.

Inicialmente sólo nuestro punto final, que debemos fijar a priori, tiene un costo determinado para *I* que hacemos igual a *0*. Para la región Ω^1 los puntos de la frontera con valores de h_2 mayores que los del punto final podrán ser de salida, ya que las trayectorias tenderán hacia el punto final. En cambio los restantes, con h_2 menor, son de retorno, ya que las trayectorias que salieran de ellos se alejarían de su objetivo. Para Ω^2 ocurre lo contrario, los puntos que para Ω^1 eran de salida son ahora de retorno y viceversa. Nos encontramos entonces con una interdependencia de las regiones, es decir, debemos de tener calculada la región Ω^2 para conocer los valores de todos los puntos finales de Ω^1 y para tener fijados los puntos de retorno de Ω^2 debemos haber calculado la región Ω^1. Para evitar esto, como hicimos en el primer sistema, asignaremos a la función de mínimo costo de los puntos finales un valor arbitrario. Con esta determinación podemos realizar el primer paso de la PD en cualquiera de las dos regiones de manera independiente.

En la figura 4.10 se presenta el esquema de los distintos elementos obtenidos para

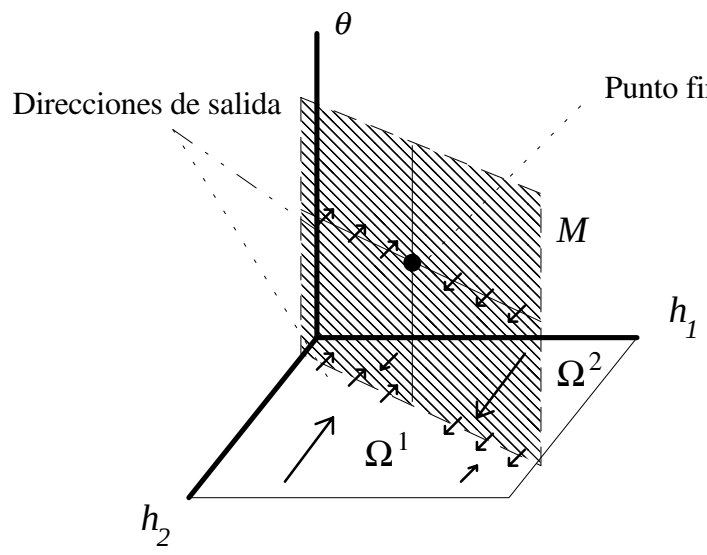

Figura 4.10 - Esquema de las Regiones del Tercer Sistema

este sistema.

4.2.3.2 Propiedad de Simetría

En este sistema nos encontramos con dos circunstancias. Por un lado, existen constantes sumadas en todas las ecuaciones, como es el caso de h_0 y $1/R_T$. Por otro lado, las alturas han de ser positivas y están limitadas por el tamaño de los depósitos. Esto hace imposible encontrar una transformación de simetría dentro del espacio de trabajo.

4.2.3.3 Expresiones del Comando y el Tiempo

Optamos por usar la aproximación $P(x)=x$ con objeto de simplificar los cálculos. Con esta consideración, las expresiones de las ecuaciones dinámicas del sistema discretizadas mediante BPF son

$$\begin{cases} 2(h_{1k+1}-h_{1k})/T_k = -R_1(\Delta h_k + \Delta h_{k+1}) + (q_{lk} + q_{lk+1}) \\ 2(h_{2k+1}-h_{2k})/T_k = R_1(\Delta h_k + \Delta h_{k+1}) - R_2(h_{2k} + h_{2k+1}) \\ 2(\theta_{k+1}-\theta_k)/T_k = \left\{-1/R_T(\theta_k+\theta_{k+1}) - C_p\left[R_1(\theta_k\Delta h_k + \theta_{k+1}\Delta h_{k+1}) + R_2(\theta_k h_{2k} + \theta_{k+1}h_{2k+1})\right]\right\}/C_s + q_{ck} + q_{ck+1} \end{cases} \tag{4.40}$$

De la segunda podemos despejar el intervalo de tiempo

$$T_k = 2\frac{h_{2k+1}-h_{2k}}{R_1(\Delta h_k + \Delta h_{k+1}) - R_2(h_{2k}+h_{2k+1})} \tag{4.41}$$

De la primera podemos obtener la expresión para q_l

$$q_{lk} = 2\frac{h_{1k+1}-h_{1k}}{T_k} + R_1(\Delta h_k + \Delta h_{k+1}) - q_{lk+1} \tag{4.42}$$

y de la tercera la de q_c

$$q_{ck} = 2\frac{\theta_{k+1}-\theta_k}{T_k} + \frac{1}{C_s}\left\{\frac{\theta_k+\theta_{k+1}}{R_T} + C_p R_1\left[\theta_k\Delta h_k + \theta_{k+1}\Delta h_{k+1}\right] - C_p R_2\left[\theta_k h_{2k} + \theta_{k+1}h_{2k+1}\right]\right\} - q_{ck+1} \tag{4.43}$$

Como en otros casos, dejamos los comandos en términos de T_k. Esto simplifica las fórmulas y no conlleva ninguna restricción ya que todas se han de calcular a la vez.

4.2.3.4 Resultados Obtenidos

En la planta el tamaño de los tanques es de 10 cm. de altura, lo cual nos da el límite de las variables h_1 y h_2. Utilizamos en las dimensiones de h_1 y h_2 100 puntos, lo que representa una discretización $d_1=d_2=0.1$. La variable θ ha de tener sólo valores positivos ya que, suministrando energía, es imposible reducir la temperatura del sistema. Elegimos

para ella una discretización $d_\theta=0.05$ y 120 puntos, dando un rango de variación en el intervalo [*0,6*]. Con esta discretización tenemos una rejilla de 100×100×120=1.200.000 puntos. Como debemos almacenar, como mínimo, 3 valores de doble precisión por punto esto significa una necesidad de memoria de aproximadamente 27Mb. Durante el proceso de cálculo, además, es preciso almacenar el costo mínimo, lo que implica una memoria total de 36Mb.

Para los parámetros tomamos unos valores de $h_0=5$, $R_1=R_2=2.8$, $R_T=1$, $C_p=1$, $C_s=1.5$, similares a los correspondientes al sistema físico. El punto final lo fijamos en $h_1^f=5$ y $\theta^f= 5$. La condición de que este punto sea estable hace que

$$\dot{h}_2^f = 0 \quad \Rightarrow \quad R_1(h_1^f - h_2^f + h_0) - R_2 h_2^f = 0 \quad \Rightarrow \quad h_1^f = (1+ R_2/R_1)h_2^f - h_0 \qquad (4.44)$$

Con el valor de los parámetros elegidos y el valor de las variables en el punto final tendremos que $h_1^f=5$. A este punto final asignamos una función de mínimo coste $I=0$, mientras que al resto de los puntos de la variedad le asignamos un valor de $I=10^6$, muy superior al del punto final, pero que permite que las trayectorias pasen de una región a otra.

En la figura 4.11 se muestran las trayectorias, para distintos puntos iniciales situados en torno al origen, cuando los comandos están limitados a *50*. Vemos que el sistema asciende hasta alcanzar el punto final. Se utiliza el caudal fuerte al principio llegándose al máximo a mitad de la trayectoria, lo que provoca que h_1 sobrepase su valor final. El suministro de energía al segundo tanque se produce en las etapas finales, no utilizándose el comando máximo. Esto provoca que baje la temperatura en aquellos casos en que se parte de una superior a la ambiente. El alza de la temperatura se consigue, coincidiendo

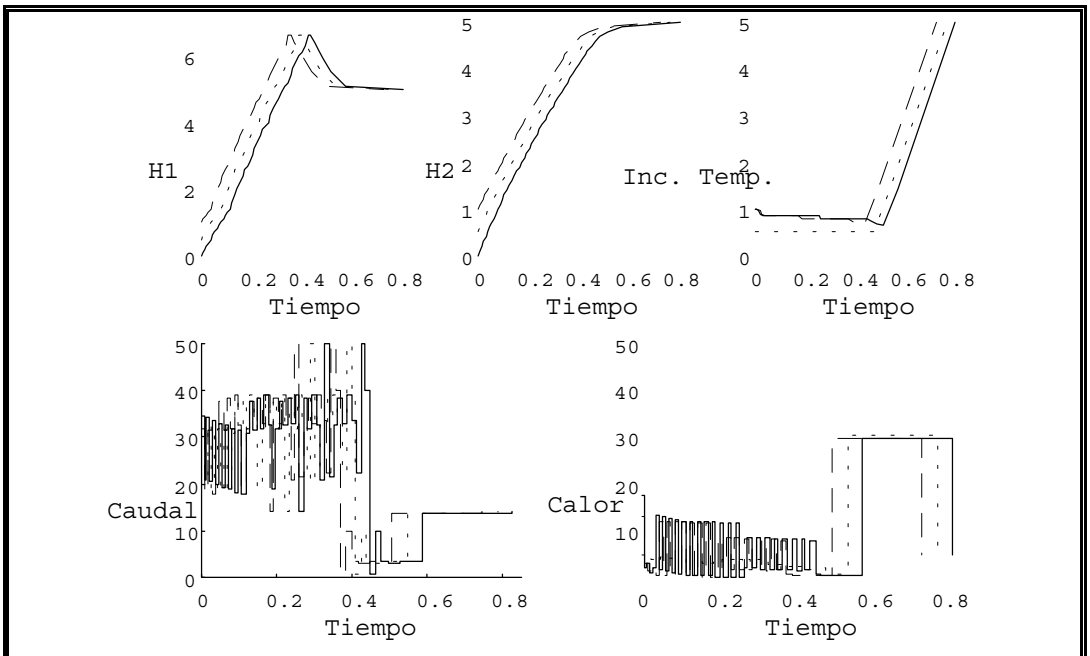

Figura 4.11 - Trayectorias para Distintos Puntos Iniciales

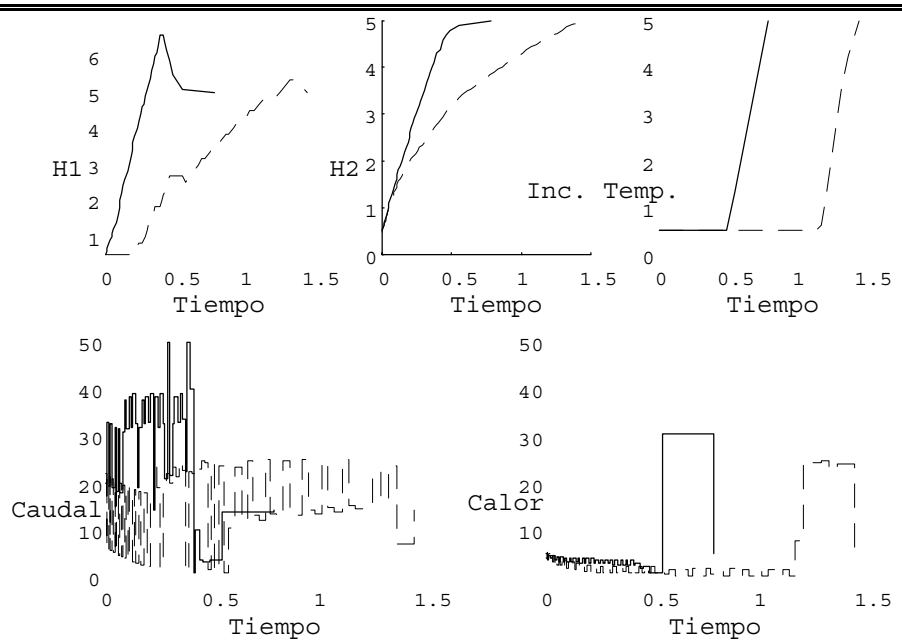

Figura 4.12 - Comparación para distintas cotas del comando

con la aplicación del comando, en las etapas finales. Para aquellos estados en que los niveles iniciales son mayores que cero el sistema tarda menos en alcanzar su objetivo.

En la figura 4.12 se comparan las trayectorias del caso anterior con una obtenida cuando la cota del caudal se ha reducido a la mitad (*25*). La forma de ambas trayectorias es muy similar pero en el segundo caso, al sólo disponer de la mitad de comando, los depósitos tardan más en llenarse y el sobrepaso del primero es menor. Como en el caso anterior el incremento de temperatura se consigue en los instantes finales.

4.2.4 Resumen y Conclusiones

Se ha mostrado la aplicación de las técnicas que hemos desarrollado para la PD a sistemas no lineales de distinto orden y complejidad, dándose situaciones diferentes en cada uno de ellos. Gracias a estas técnicas se han podido calcular trayectorias de tiempo mínimo con una importante reducción de la complejidad, tanto espacial como temporal, con respecto a la PD clásica. A pesar de ello, la memoria requerida para una discretización suficiente crece muy rápidamente con la dimensión del sistema. Desde los 476Kb del primer ejemplo pasamos a necesitar 34Mb y 36Mb en el segundo y tercero respectivamente. Esto hace difícil la aplicación real de esta política a sistemas de tercer orden o superior.

5

Primer Método Basado en Redes Neuronales

En este capítulo presentamos el primero de los métodos de aplicación de redes neuronales al control óptimo. Comenzamos exponiendo la idea de la que éste surge a partir de la PD. En la siguiente sección lo describimos en detalle desarrollando las fórmulas necesarias para el entrenamiento de las redes. Finalmente, al igual que hacemos con los otros métodos, presentamos los resultados obtenidos sobre nuestros sistemas ejemplo.

5.1 RN como Alternativa a la PD

En el capítulo anterior se ha mostrado la aplicación de la PD para resolver PCOG. A pesar de las técnicas aplicadas, la complejidad sigue siendo alta. Es necesario almacenar para el conjunto discreto de estados, o al menos para la mitad de ellos, el intervalo de tiempo y el comando a aplicar. En una aplicación real, para evitar la necesidad de interpolar en el segundo paso, la discretización ha de ser igual a la que proporciona la resolución de los conversores A/D.

Como mencionamos en el apartado 1.3.5, una forma subóptima de solucionar el problema de la complejidad en aplicaciones reales es conseguir una función que aproxime la política de control óptimo. Una red neuronal podría implementar esa función en cada etapa. Esta red podría incluso dar los valores para todo un rango continuo, aunque sea entrenada con los puntos de la discretización, debido a la capacidad de generalizar que tienen las RN. La figura 5.1 muestra el esquema de esta solución.

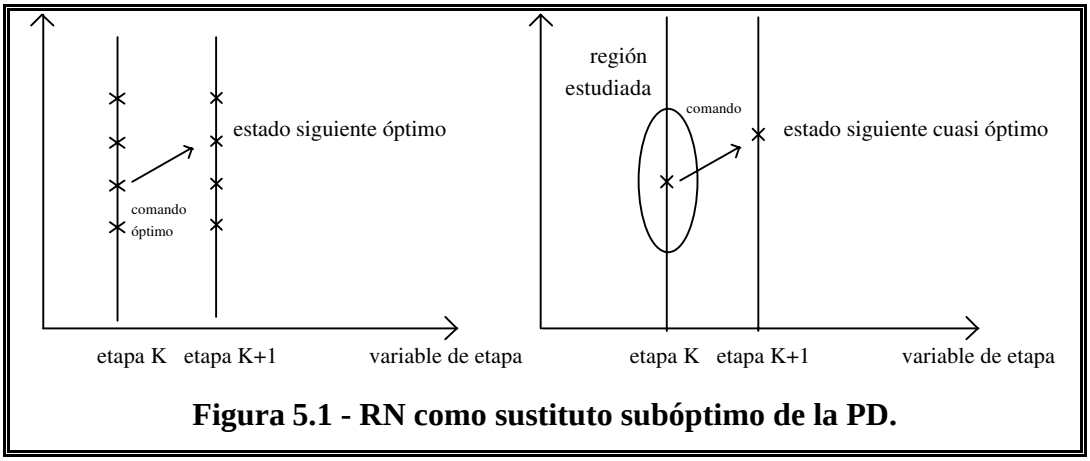

Figura 5.1 - RN como sustituto subóptimo de la PD.

Una aplicación directa sería, entonces, entrenar una RN con parejas de entrenamiento *(estado,comando)* obtenidas por la PD para una etapa. Esta solución, aunque directa, no reduciría mucho el problema de la complejidad debido al gran número de etapas que se han de utilizar en la PD para obtener un buen resultado. La complejidad espacial resultante será todavía grande y, además, se aumentaría la complejidad temporal por el proceso de entrenamiento de las redes.

Una solución derivada de ésta es entrenar cada red con los resultados de varias etapas, posiblemente utilizando el valor de la etapa como una entrada. Esta solución reduciría la complejidad espacial y temporal. En cambio aparece otro problema: la elección de las etapas que le corresponden a cada red. De esta elección dependerá, en gran medida, la optimalidad de la solución.

Un problema común a estos procedimientos es que las variaciones, dentro de una etapa, del resto de las variables de estado puede ser muy grande. Esto puede hacer necesario una red grande para que sea capaz de aprender todas las variaciones. Si la red no se elige suficientemente buena la solución puede ser muy pobre.

El método de solución aquí propuesto consiste en permitir a las redes que, en el proceso de minimización, se coloquen, por así decirlo, en la etapa que más convenga. En realidad lo que ocurre es que no se define tal variable de etapa. Las RN forman una estructura de cadena. Dado el estado inicial éste será la entrada a la RN de la primera etapa. Esta red nos dará, a su salida, el valor del comando e intervalo de tiempo que,

aplicado a las ecuaciones discretizadas del sistema, generarán el estado de la segunda etapa. Este estado será introducido a la RN de la segunda etapa para producir el segundo comando e intervalo de tiempo, y así sucesivamente.

Previamente se determina cual es la región a la que van a pertenecer los estados iniciales. Para las siguientes etapas no se conocen, a priori, cuales van a ser sus regiones de entrada o salida, sino que es el propio proceso de minimización el que las determina. De esta manera las redes sustituyen de forma óptima a una serie de etapas de la PD. Además, se cubre una determinada región de cada etapa y no todo el rango posible en cada una de ellas. De esta manera, al contrario de lo que ocurría en los planteamientos anteriores, las redes pueden ser más sencillas.

La cadena de RN debe ser entrenada para generar la trayectoria óptima. La determinación de esta trayectoria se hace basándose únicamente en la función de coste y no en los resultados de la PD calculada previamente. El algoritmo de entrenamiento más conveniente, como veremos, es la BP. Este algoritmo es, en esencia, un sustituto del primer barrido de la PD. Los errores se van propagando desde las etapas finales hacia las iniciales de forma análoga a como se define el costo mínimo de una etapa en función del de las siguientes en la PD. Por la forma en que se realiza el entrenamiento, cuanto mayor sea el número de pares de entrenamiento, al minimizar en conjunto, menor será la optimización particular. En nuestro caso esto se traduce en que cuanto mayor es la región inicial de entrenamiento menor es la optimalidad de las trayectorias para cada uno de sus puntos.

Como las regiones de entrada y salida de las redes, salvo para la primera, no están determinadas, no se puede fijar un punto o región final. Si esto es lo que deseamos, debemos incluir un costo en la etapa final conveniente para que en el proceso de entrenamiento las trayectorias alcancen la región final deseada.

En [Nguy 90] y [Pari 94] se han propuesto estructuras similares pero sus redes sólo generan el comando, por lo tanto no pueden resolver problemas de tiempo mínimo. Existen, támbien, otras aplicaciones de las redes neuronales al control óptimo [Nare 90] [Nare 91] [Werb 91].

5.2 Descripción del Método

Este método consiste en dividir la trayectoria en N etapas, no ligadas a ninguna variable. En cada etapa se coloca una red neuronal que aprenderá el comando e intervalo de tiempo óptimos para los estados de esa etapa [Hami 92] [More 93b] [Acos 94] [Hami 95]. Si denominamos γ a la función implementada por cada red tendremos que

$$\underline{v}_i \equiv (\underline{u}_i, T_i) = \gamma(\underline{w}_i, \underline{x}_i) \tag{5.1}$$

donde por $\underline{w}_i$ representamos el vector de todos los pesos de la red *i*-ésima. Aplicando este comando e intervalo de tiempo al sistema obtendremos el estado en la etapa siguiente. El esquema completo es el presentado en la figura 5.2.

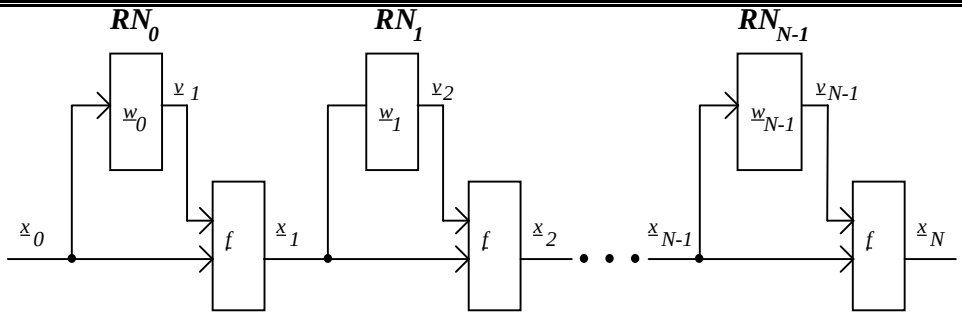

Figura 5.2 - Esquema del Primer Método

La ecuación dinámica discretizada del sistema se pondrá como

$$\underline{x}_{i+1} = \underline{f}(\underline{x}_i, \underline{u}_i, T_i, i) = \underline{f}(\underline{x}_i, \underline{v}_i, i) \tag{5.2}$$

Si los puntos iniciales pertenecen a la región $A^0 \subset \mathbb{R}^n$, los estados de las sucesivas etapas pertenecerán a las regiones A^i, con $i=1,\ldots,N$.

El tipo de red considerada es el de perceptrón multicapa. Este tipo de redes pueden implementar perfectamente las funciones continuas que se desea. Eligiendo convenientemente las funciones de activación de la última capa y su escalado podemos implementar las restricciones en los comandos. Por ejemplo, si para un comando se desea un rango de [*-3,3*] colocaremos, en la neurona que dé la salida correspondiente, una función tipo tangente hiperbólica multiplicada por un factor *3*. De forma similar se puede hacer que la salida sea sólo positiva o sencillamente ilimitada. Esta flexibilidad es la que nos permite tratar con intervalos de tiempo variables. Nos basta con colocar una función tipo sigmoide, de salida [*0,1*], y escalarla para que cubra un rango razonable en el intervalo de tiempo. A pesar de estas consideraciones, el desarrollo que se presenta a continuación no las tendrá en cuenta y se hará para una función de activación cualquiera. La única condición que se le exigirá es que posea derivada primera.

En cuanto a las restricciones en el estado, como pueden ser un punto final fijo o zonas prohibidas para el sistema, han de implementarse en la función de costo. Tomaremos la función de costo siguiente

$$J = S(\underline{x}_N, N) + \sum_{i=0}^{N-1} L(\underline{x}_i, \underline{v}_i, i) \tag{5.3}$$

Para el caso del estado final, basta con pesar fuertemente en la función S aquellos estados que estén fuera del conjunto de estados deseados, de manera que el sistema tienda hacia dicho conjunto. Para las zonas prohibidas se aplica la idea contraria. Se han de pesar en L los estados que caigan dentro de dicha zona y de manera que el sistema tienda a salir de ella, es decir, haciendo que el costo disminuya a medida que el estado se acerque al frontera.

5.2.1 Cálculo de las Trayectorias

Como vamos a trabajar con varias redes utilizaremos un superíndice en todos los elementos para indicar la red a la que pertenecen. Por esto la ecuación de una neurona de la red *i*-ésima será

$$z_q^i(s) = \sum_{p=1}^{n_{s-1}^i} w_{pq}^i(s) y_p^i(s-1) + w_{0q}^i(s); \quad y_q^i(s) = g(z_q^i(s)) \tag{5.4}$$

donde $s=1,...,L^i$ es el número de la capa y $q=1,...,n_s^i$ es el cardinal de la neurona dentro de la capa s. Podemos agrupar como vector al conjunto de salidas de la capa s:

$$\underline{y}^i(s) = \left(y_1^i(s), y_2^i(s), \ldots, y_{n_s}^i(s)\right)^{\mathrm{T}} \tag{5.5}$$

y también al conjunto de pesos de entrada, sin contar el de sesgo, de la neurona q de la capa s:

$$\underline{w}_q^i(s) = \left(w_{1q}^i(s), w_{2q}^i(s), \ldots, w_{n_s q}^i(s)\right)^{\mathrm{T}} \tag{5.6}$$

De esta manera, la ecuación anterior se podrá poner como

$$z_q^i(s) = \underline{w}_q^i(s)^{\mathrm{T}}\, \underline{y}^i(s-1) + w_{0q}^i(s); \quad y_q^i(s) = g(z_q^i(s)) \tag{5.7}$$

El proceso de cálculo de una trayectoria completa será el siguiente

```
Elegir el estado inicial x_0
Para i=0 hasta N-1
   Tomar y_i(0)=x_i
   Para s=1 hasta L^i
      aplicar (5.7) para obtener y_i(s)
   Tomar v_i=y_i(L^i)
   Obtener x_{i+1} aplicando (5.2)
```

Por este algoritmo determinamos tanto la secuencia de comandos e intervalos de tiempo aplicados $\{\underline{v}_0, \underline{v}_1, \ldots, \underline{v}_{N-1}\}$ como la trayectoria $\{\underline{x}_0, \underline{x}_1, \ldots, \underline{x}_{N-1}\}$.

5.2.2 Proceso de Entrenamiento

Nuestro objetivo es entrenar esta cadena de redes de manera que se minimice el costo de la trayectoria para los puntos iniciales pertenecientes a la región inicial A^0. Dado un vector de entrada no sabemos, a priori, cuales han de ser los vectores de salida de cada red. Es decir, no existen las parejas *(vector de entrada, vectores de salida deseada)* típicas de un entrenamiento supervisado. Podría pensarse, entonces, que nos encontramos frente a un caso de entrenamiento no supervisado. Lo que ocurre realmente es que nosotros no conocemos los vectores de salida deseados. Los podríamos conocer si resolvemos el problema de control óptimo, por ejemplo aplicando la PD. De lo que disponemos ahora, en vez de esos vectores deseados, es directamente de la función de error: la función de costo de la trayectoria. En realidad los vectores de salida deseados se utilizaban, en el caso general, sencillamente para construir dicha función. Por usar como función de error la función de costo, a la vez que entrenamos las redes, estamos encontrando la solución del PCO.

La forma en que son elegidos los vectores de entrenamiento determina la optimización que se realiza. Si deseamos una optimización global para todos los puntos de la región inicial, los vectores de entrada han de ser elegidos de una distribución uniforme sobre dicha región. También es posible elegir otro tipo de distribución, consiguiendo de esa manera mayor optimización para ciertos puntos de la región. Habitualmente este conjunto de vectores de entrenamiento no es fijo, sino que en cada paso de entrenamiento se elige aleatoriamente uno nuevo.

5.2.3 Fórmulas para la Backpropagation

A la vista de estas consideraciones, el algoritmo de entrenamiento a utilizar es la BP, ya que es adecuado al tipo de red y a nuestros propósitos de minimización. La aplicación de la BP, en cualquiera de sus variantes, precisa el conocimiento del gradiente de la función de costo J con respecto a los pesos. Para cada peso, al igual que en el caso general descrito en la sección 2.5.6, escribimos que

$$\frac{\partial J}{\partial w_{pq}^{i}(s)} = \delta_q^i(s) y_p^i(s-1) \tag{5.8}$$

donde habíamos definimos la función δ para la neurona q de la capa s como

$$\delta_q^i(s) \equiv \frac{\partial J}{\partial z_q^i(s)} \tag{5.9}$$

El algoritmo ha de ser extendido, ya que el error ha de ser propagado, no sólo dentro de una red, sino de una red a otra. Además existen componentes del error que se generan

en cada etapa. Para aclarar esto en la figura 5.3 presentamos el esquema de la cadena añadiendo el diagrama de generación del costo. Como se ve, no sólo existe el costo J^N debido al punto final y generado por la función *S*, sino que en cada etapa se genera el costo J^i calculado mediante la función *L*. Esto hace necesario modificar las fórmulas para la obtención de δ.

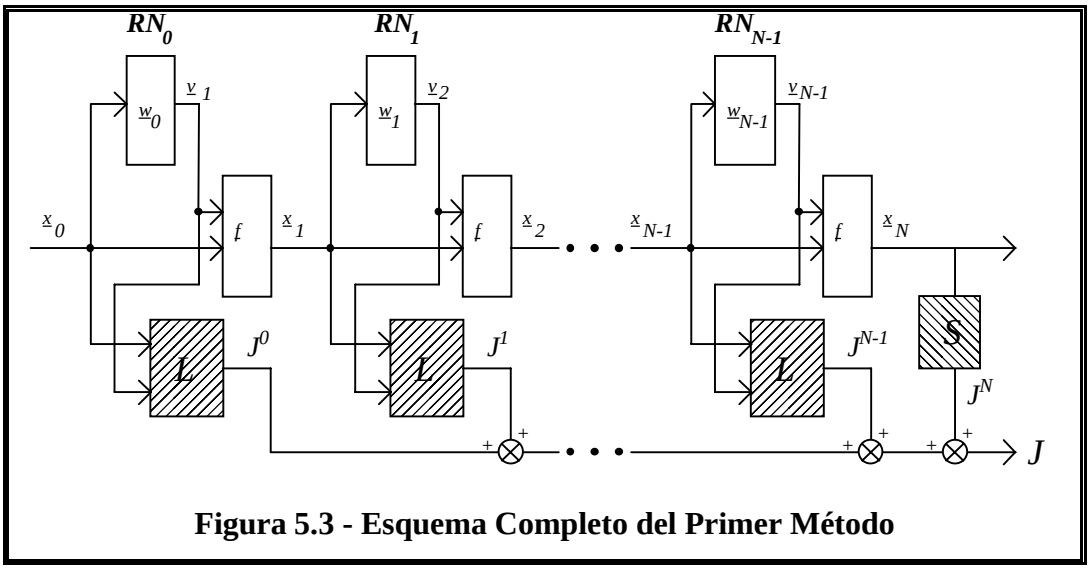

Figura 5.3 - Esquema Completo del Primer Método

La forma de obtener δ para las capas ocultas es la misma que en el caso general salvo que se indica la red en el superíndice

$$\delta_q^i(s) = \frac{\partial J}{\partial z_q^i(s)} = \frac{\partial J}{\partial y_q^i(s)} \frac{\partial y_q^i(s)}{\partial z_q^i(s)} = \left[\sum_{h=1}^{n_{s+1}^i} \delta_h^i(s+1) w_{qh}^i(s+1) \right] g'(z_q^i(s)) \tag{5.10}$$

Para la capa de salida, en cambio, la expresión ha de cambiar. Repitiendo el desarrollo teníamos que

$$\delta_q^i(L^i) = \frac{\partial J}{\partial z_q^i(L^i)} = \frac{\partial J}{\partial y_q^i(L^i)} \frac{\partial y_q^i(L^i)}{\partial z_q^i(L^i)} = \left[\frac{\partial J}{\partial v_q^i} \right] g'(z_q^i(L^i)) \tag{5.11}$$

Donde hemos usado que $y_q^i(L^i) = [\underline{v}_i]_q$.La expresión de la parcial del costo con respecto al vector de salida se descompondrá en

$$\frac{\partial J}{\partial \underline{v}_i} = \frac{\partial L(\underline{x}_i, \underline{v}_i, i)}{\partial \underline{v}_i} + \frac{\partial J}{\partial \underline{x}_{i+1}} \frac{\partial \underline{x}_{i+1}}{\partial \underline{v}_i} \tag{5.12}$$

El primer sumando es debido a la aportación que hace el vector $\underline{v}_i$ al coste directamente a través de la función *L*. El segundo representa la aportación que hace a través de la generación del estado siguiente $\underline{x}_{i+1}$. Por conveniencia vamos a definir el vector

$$\underline{\lambda}_i \equiv \nabla_{\underline{x}_i} J = \left(\frac{\partial J}{\partial \underline{x}_i} \right)^{\mathrm{T}} \tag{5.13}$$

que, como vemos, está asociado a cada red. Con esta definición la expresión anterior queda

$$\frac{\partial J}{\partial \underline{v}_i} = \frac{\partial L(\underline{x}_i, \underline{v}_i, i)}{\partial \underline{v}_i} + \underline{\lambda}_i{}^{\mathrm{T}} \frac{\partial \underline{x}_{i+1}}{\partial \underline{v}_i} \tag{5.14}$$

Llevando este resultado a (5.11) obtendremos la expresión de los δ de la última capa en función de este nuevo vector $\underline{\lambda}$:

$$\delta_q^i(L^i) = \left[\frac{\partial}{\partial \underline{v}_i} L(\underline{x}_i, \underline{v}_i, i) + \underline{\lambda}_{i+1}^{\mathrm{T}} \frac{\partial}{\partial \underline{v}_i} \underline{f}(\underline{x}_i, \underline{v}_i) \right]_q g'(z_q^i(L^i)) \tag{5.15}$$

expresión válida para todas las redes, es decir, para *i=0,1,…,N-1*.

Debemos ahora encontrar la expresión para $\underline{\lambda}_i$. Para ello descomponemos el gradiente en sus componentes aplicando la regla de la cadena. En el caso general (*i=0,…,N-1*) tendremos que

$$\underline{\lambda}_i^{\mathrm{T}} = \frac{\partial J}{\partial \underline{x}_i} = \frac{\partial L(\underline{x}_i, \underline{v}_i, i)}{\partial \underline{x}_i} + \frac{\partial J}{\partial \underline{x}_{i+1}} \frac{\partial \underline{x}_{i+1}}{\partial \underline{x}_i} + \frac{\partial J}{\partial \underline{y}^i(0)} \frac{\partial \underline{y}^i(0)}{\partial \underline{x}_i} \tag{5.16}$$

donde el primer sumando es la aportación de $\underline{x}_i$ al costo en esa etapa; el segundo es la aportación al generar $\underline{x}_{i+1}$; y el último por ser entrada a la red *i*-ésima. Aplicando la definición de $\underline{\lambda}$ al segundo de esos sumandos y teniendo en cuenta que $\underline{x}_{i+1}=\underline{f}(\underline{x}_i,\underline{v}_i,i)$, se podrán poner como

$$\frac{\partial J}{\partial \underline{x}_{i+1}} \frac{\partial \underline{x}_{i+1}}{\partial \underline{x}_i} = \underline{\lambda}_{i+1}^{\mathrm{T}} \frac{\partial \underline{f}(\underline{x}_i, \underline{v}_i, i)}{\partial \underline{x}_i} \tag{5.17}$$

Para el tercer sumando debemos tener en cuenta que $\underline{y}_i(0)=\underline{x}_i$ y aplicar la regla de la cadena dentro de la neurona

$$\frac{\partial J}{\partial \underline{y}^i(0)} \frac{\partial \underline{y}^i(0)}{\partial \underline{x}_i} = \left[\sum_{q=1}^{n} \frac{\partial J}{\partial z_q^i(1)} \frac{\partial z_q^i(1)}{\partial \underline{y}^i(0)} \right] \cdot 1 \tag{5.18}$$

Por la expresión (5.7) y reconociendo la definición de δ, podremos poner

$$\frac{\partial J}{\partial \underline{y}^i(0)} \frac{\partial \underline{y}^i(0)}{\partial \underline{x}_i} = \sum_{q=1}^{n} \delta_q^i(1) \underline{w}_q^i(1)^{\mathrm{T}} \tag{5.19}$$

Llevando estos resultados a (5.16) tendremos que

$$\underline{\lambda}_i^{\mathrm{T}} = \frac{\partial L(\underline{x}_i, \underline{v}_i, i)}{\partial \underline{x}_i} + \underline{\lambda}_{i+1}^{\mathrm{T}} \frac{\partial \underline{f}(\underline{x}_i, \underline{v}_i, i)}{\partial \underline{x}_i} + \sum_{q=1}^{n} \delta_q^i(1) \underline{w}_q^i(1)^{\mathrm{T}} \tag{5.20}$$

lo que representa una expresión recursiva en $\underline{\lambda}$, válida para $i=1,\ldots,N-1$. El caso base lo determinamos estudiando la expresión de lambda para $i=N$. Ésta será, directamente de la definición,

$$\underline{\lambda}_N = \frac{\partial S(\underline{x}_N, N)}{\partial \underline{x}_N} \tag{5.21}$$

De esta manera tenemos determinadas todas las variables del proceso y podemos plantear el algoritmo para un paso de entrenamiento:

```
Elegir un estado inicial x_0 ∈ A^0
Calcular la trayectoria para ese estado inicial
Determinar λ_N mediante (5.21)
Para i=N-1 hasta 0
   Calcular δ de la última capa mediante (5.15)
   Para s=L^i-1 hasta 1
      Calcular δ de la capa s mediante (5.10)
   Calcular λ_i mediante (5.20)
Determinar la parcial de J con respecto a cada
peso mediante (5.8)
Aplicar la corrección de los pesos según el
algoritmo elegido
```

Hacer notar que las operaciones que se indican en una línea, aunque implican a varios elementos, como por ejemplo calcular los delta de toda una capa, son independientes entre si y por lo tanto podrían hacerse en paralelo, si la implementación lo permitiera.

Un aspecto importante es que, para aplicar este algoritmo, no sólo es necesario poseer el modelo dinámico de forma (5.2) sino que es preciso conocer sus derivadas con respecto a todos sus argumentos. De la misma manera, es necesario conocer las derivadas parciales de las funciones de costo L y S con respecto a las variables de estado y los comandos. Estas derivadas han de calcularse de forma analítica o al menos que puedan evaluarse con facilidad, ya que han de usarse repetidas veces durante cada paso de entrenamiento. El proceso de cálculo de estas derivadas puede ser complejo si lo es la expresión discretizada de las ecuaciones del sistema. Por otro lado solamente hay que calcularlas cuando se fija el modelo del sistema y siguen siendo válidas ante cambios de los parámetros.

5.2.4 Estudio de la Complejidad

La complejidad de los algoritmos de cálculo y aprendizaje es del orden del número de pesos. Es necesario, en el proceso de cálculo, multiplicar cada peso por su entrada,

mientras que en el proceso de entrenamiento debemos calcular la derivada del costo con respecto a cada peso.

Como vemos por las fórmulas, tanto de cálculo de la trayectoria como del entrenamiento, las dimensiones del sistema se reflejan en el número de entradas y salidas de las redes. Cada red tendrá un número de entradas igual a la dimensión del vector de estados, es decir, igual a la dimensión del sistema. El número de salidas ha de ser igual al número de comandos más una salida que proporciona el intervalo de tiempo.

Por esta característica, el tratar con un sistema de un orden superior representa añadir una entrada más a las neuronas de la primera capa. Si tenemos redes con $n_1^i=n_1$ neuronas en la primera capa, esta nueva entrada supondrá un aumento de $n_1 \cdot N$ pesos. Es decir, si tenemos una estructura fija, el tratar con un sistema de dimensión n supone aumentar el número de pesos en $n \cdot n_1 \cdot N$. Como vemos se trata de un aumento lineal con la dimensión del sistema.

Aumentar el número de comandos implica colocar una neurona más en la capa de salida. Si tenemos una estructura de redes de n_L neuronas en la capa penúltima, añadir una neurona implica aumentar en $(n_L+1) \cdot N$ pesos. Como antes, si partimos de una estructura fija y queremos tratar un sistema de m comandos, supone una aumento de $(m+1) \cdot (n_L+1) \cdot N$ pesos. Se trata, nuevamente, de un aumento de complejidad lineal con la dimensión del comando.

Esta dependencia lineal del número de pesos con la dimensión del sistema implica aumento lineal de complejidad. Ésta es una gran ventaja de este método con respecto a la solución mediante PD que, como vimos, presenta complejidad exponencial.

5.3 Aplicación a los Problemas Ejemplo

Para poder estudiar el comportamiento de este método lo aplicaremos a nuestros sistemas ejemplo.

5.3.1 Primer Sistema

5.3.1.1 Expresiones Necesarias

Como indicamos, para el proceso de entrenamiento es necesario conocer el modelo discretizado del sistema y sus derivadas con respecto a los estados y comandos. Las ecuaciones dinámicas, como indica (5.2), han de obtener el estado siguiente en función del estado actual, el comando aplicado y el intervalo de tiempo. Expresiones de este tipo

sólo pueden obtenerse aplicando la aproximación de la diferencia adelante y no otras como diferencia atrás o BPF.

La ecuaciones discretizadas por diferencia adelante para este sistema eran

$$\begin{cases} (x_{1k+1} - x_{1k})/T_k = x_{2k} \\ (x_{2k+1} - x_{2k})/T_k = ax_{2k} + bx_{2k}{x_{1k}}^2 + dx_{1k} + cu_k \end{cases} \tag{3.30}$$

de ésta podemos despejar

$$\begin{cases} x_{1k+1} = x_{1k} + T_k x_{2k} \\ x_{2k+1} = x_{2k} + T_k\left[ax_{2k} + bx_{2k}{x_{1k}}^2 + dx_{1k} + cu_k\right] \end{cases} \tag{5.22}$$

Las derivadas del tipo $\partial \underline{f} / \partial \underline{x}_k$ son las siguientes:

$$\frac{\partial f_1}{\partial x_{1k}} = \frac{\partial x_{1k+1}}{\partial x_{1k}} = 1 \tag{5.23}$$

$$\frac{\partial f_1}{\partial x_{2k}} = \frac{\partial x_{1k+1}}{\partial x_{2k}} = T_k \tag{5.24}$$

$$\frac{\partial f_2}{\partial x_{1k}} = \frac{\partial x_{2k+1}}{\partial x_{1k}} = T_k\left[2bx_{2k}x_{1k} + d\right] \tag{5.25}$$

$$\frac{\partial f_2}{\partial x_{2k}} = \frac{\partial x_{2k+1}}{\partial x_{2k}} = 1 + T_k\left[a + b{x_{1k}}^2\right] \tag{5.26}$$

Las derivadas del tipo $\partial \underline{f} / \partial \underline{v}_k$ son las siguientes:

$$\frac{\partial f_1}{\partial T_k} = \frac{\partial x_{1k+1}}{\partial T_k} = x_{2k} \tag{5.27}$$

$$\frac{\partial f_1}{\partial u_k} = \frac{\partial x_{1k+1}}{\partial u_k} = 0 \tag{5.28}$$

$$\frac{\partial f_2}{\partial T_k} = \frac{\partial x_{2k+1}}{\partial T_k} = \left[ax_{2k} + bx_{2k}{x_{1k}}^2 + dx_{1k} + cu_k\right] \tag{5.29}$$

$$\frac{\partial f_2}{\partial u_k} = \frac{\partial x_{2k+1}}{\partial u_k} = cT_k \tag{5.30}$$

Queríamos obtener las trayectorias de tiempo mínimo por lo que definimos

$$L(\underline{x}_k, \underline{v}_k, k) = T_k \tag{5.31}$$

Para obtener como punto final el origen debíamos pesar fuertemente la distancia a dicho punto en la etapa final. Esto lo hacemos mediante la función *S*

$$S(\underline{x}_N, N) = V_N \cdot (x_{1N}{}^2 + x_{2N}{}^2) \tag{5.32}$$

donde V_N es grande.

Las derivadas necesarias de estas funciones son

$$\frac{\partial L}{\partial \underline{x}_k} = \left(\frac{\partial L}{\partial x_{1k}} \quad \frac{\partial L}{\partial x_{2k}} \right) = \begin{pmatrix} 0 & 0 \end{pmatrix} \tag{5.33}$$

$$\frac{\partial L}{\partial \underline{v}_k} = \left(\frac{\partial L}{\partial u_k} \quad \frac{\partial L}{\partial T_k} \right) = \begin{pmatrix} 0 & 1 \end{pmatrix} \tag{5.34}$$

$$\frac{\partial S}{\partial \underline{x}_N} = \left(\frac{\partial S}{\partial x_{1N}} \quad \frac{\partial S}{\partial x_{2N}} \right) = \begin{pmatrix} 2V_N x_{1N} & 2V_N x_{2N} \end{pmatrix} \tag{5.35}$$

5.3.1.2 Resultados Obtenidos

Fijamos los parámetros del sistema al mismo valor que en el caso de la PD para poder comparar los resultados. Para implementar la restricción en el comando elegimos como función de activación la tangente hiperbólica para la primera neurona de salida. Para el

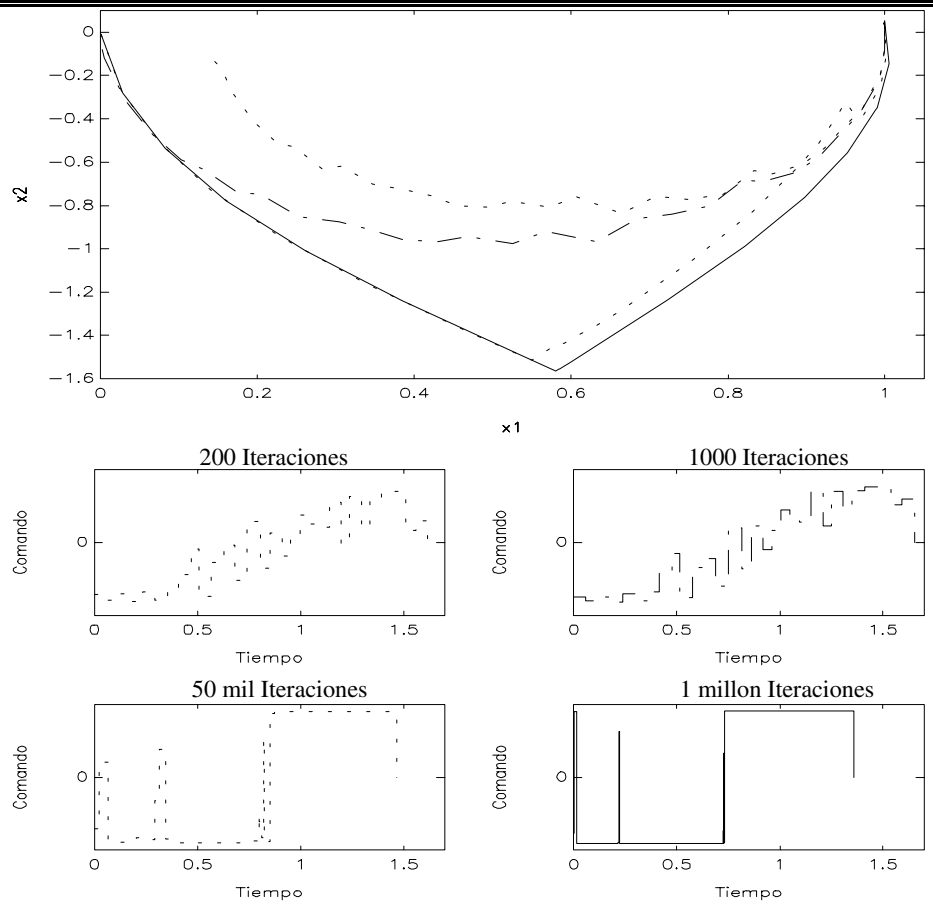

Figura 5.4 - Evolución del entrenamiento para región de único punto.

intervalo de tiempo colocamos un rango de *0.1*, escalando por ese factor la función de activación sigmoide de la segunda salida. Utilizamos 20 etapas y una estructura de las redes de dos capas ocultas de 5 neuronas cada una.

Probando con el algoritmo básico de la BP con un factor de aprendizaje $\eta=10^{-4}$ y un momento de $\alpha=0.8$ la cadena se entrenaba rápidamente. En un millón de iteraciones se llega a una solución aceptable.

Como hemos indicado, la optimalidad de la trayectoria depende de la región inicial entrenada. Para el caso de $A^0=\{(1,0)\}$, un único punto, la evolución de la trayectoria es rápida llegando a una forma casi perfecta (figura 5.4). El costo de esta trayectoria es muy cercano al obtenido mediante PD. Cuando el tamaño de la región inicial aumenta, $A^0=\{(x_1,x_2):0.75\leq x_1\leq 1.25, x_2=0\}$, las trayectorias no son tan buenas para todos los puntos (figura 5.5) y el costo aumenta. La comparación de estos valores se hace en la tabla **5.1**.

	COSTO		
Punto Inicial	PD	A^0 único punto	A^0 Segmento
(1,0)	1.2689	1.3667	1.5638

Tabla 5.1 Comparación de Costos Según la Región A^0

Como hemos visto, cuanto menor es la región mejores son los resultados, pero cuando presentamos un punto inicial no perteneciente a la región entrenada la trayectoria

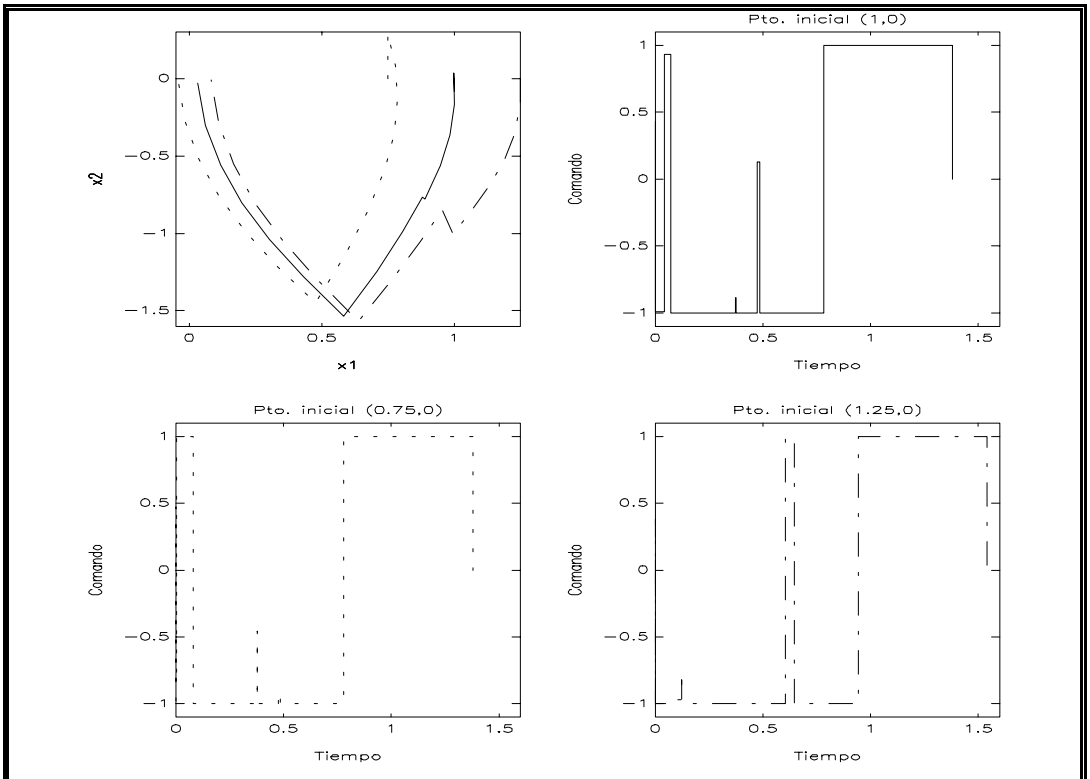

Figura 5.5 - Trayectorias para región $A^0=\{(x_1,x_2):0.75\leq x_1\leq 1.25, x_2=0\}$

calculada está muy lejos del óptimo. En la figura 5.6 presentamos el caso de dos puntos fuera de la región A^0. Como se ve, la forma de la trayectoria se reproduce pero, como se parte de un punto inicial distinto, el punto final está muy alejado del deseado. Esto supone un importante inconveniente en su utilización en un sistema real ya que cualquier perturbación que saque al sistema de cualquiera de las regiones A^i hará que la trayectoria se desvíe mucho de la óptima.

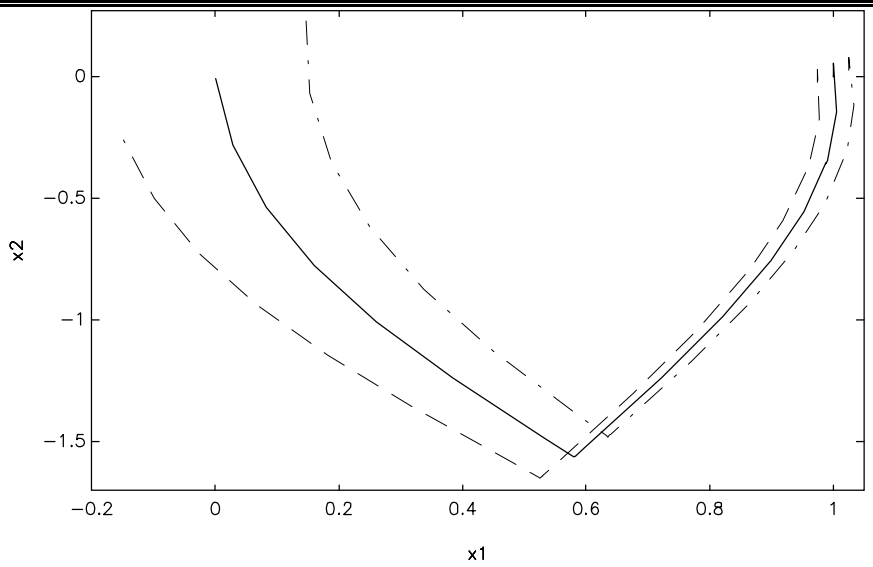

Figura 5.6 - Perturbación del estado inicial.

5.3.2 Segundo Sistema

5.3.2.1 Expresiones Necesarias

Como en el problema anterior, la única discretización que podemos usar es la de diferencia adelante por la necesidad de despejar el estado siguiente en función del estado y comandos actuales. Con esta discretización las ecuaciones serán

$$\begin{cases} x_{1k+1} = x_{1k} + T_k \cos(x_{3k}) u_{1k} \\ x_{2k+1} = x_{2k} + T_k \left[a + \text{sen}(x_{3k}) u_{1k} \right] \\ x_{3k+1} = x_{3k} + T_k u_{2k} \end{cases} \tag{5.36}$$

Para aplicar las fórmulas del entrenamiento debemos calcular las distintas derivadas parciales. Las derivadas del tipo $\partial \underline{f} / \partial \underline{x}_k$ son las siguientes:

$$\frac{\partial \underline{x}_{k+1}}{\partial x_{1k}} = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^{\mathrm{T}} \tag{5.37}$$

$$\frac{\partial \underline{x}_{k+1}}{\partial x_{2k}} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}^{\mathrm{T}} \tag{5.38}$$

$$\frac{\partial \underline{x}_{k+1}}{\partial x_{3k}} = \begin{pmatrix} -T_k \operatorname{sen}(x_{3k}) u_{1k} & T_k \left[a + \cos(x_{3k}) u_{1k} \right] & 1 \end{pmatrix}^{\mathrm{T}} \tag{5.39}$$

Las derivadas del tipo $\partial \underline{f} / \partial \underline{v}_k$ son las siguientes:

$$\frac{\partial \underline{x}_{k+1}}{\partial u_{1k}} = \begin{pmatrix} T_k \cos(x_{3k}) & T_k \operatorname{sen}(x_{3k}) & 0 \end{pmatrix}^{\mathrm{T}} \tag{5.40}$$

$$\frac{\partial \underline{x}_{k+1}}{\partial u_{2k}} = \begin{pmatrix} 0 & 0 & T_k \end{pmatrix}^{\mathrm{T}} \tag{5.41}$$

$$\frac{\partial \underline{x}_{k+1}}{\partial T_k} = \begin{pmatrix} \cos(x_{3k}) u_{1k} & a + \operatorname{sen}(x_{3k}) u_{1k} & u_{2k} \end{pmatrix}^{\mathrm{T}} \tag{5.42}$$

Deseamos trayectorias de tiempo mínimo por lo que definimos

$$L(\underline{x}_k, \underline{v}_k, k) = T_k \tag{5.43}$$

Sus derivadas son, por lo tanto

$$\frac{\partial L}{\partial \underline{x}_k} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \tag{5.44}$$

$$\frac{\partial L}{\partial \underline{v}_k} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{5.45}$$

Como en el caso de PD, deseamos que el sistema llegue al origen pero con cualquier orientación, por ello sólo pesaremos las variables x_1 y x_2, mientras que x_3 quedará libre.

$$S(\underline{x}_N, N) = V_N \cdot (x_{1N}{}^2 + x_{2N}{}^2) \tag{5.46}$$

donde, como antes, V_N ha de ser grande para que esta condición pese más que la minimización del tiempo. Sus derivadas son

$$\frac{\partial S}{\partial \underline{x}_N} = \begin{pmatrix} 2V_N x_{1N} & 2V_N x_{2N} & 0 \end{pmatrix} \tag{5.47}$$

5.3.2.2 Resultados

Como en el estudio mediante PD, fijamos un valor para el parámetro *a=0.01*. Tomamos *N*=15 con redes iguales de una capa oculta de 5 neuronas. La capa de salida de tres neuronas: la primera sigmoide escalada por *10*, ya que estamos suponiendo sólo avance y no retroceso; la segunda una tangente hiperbólica escalada por *5*, ya que se

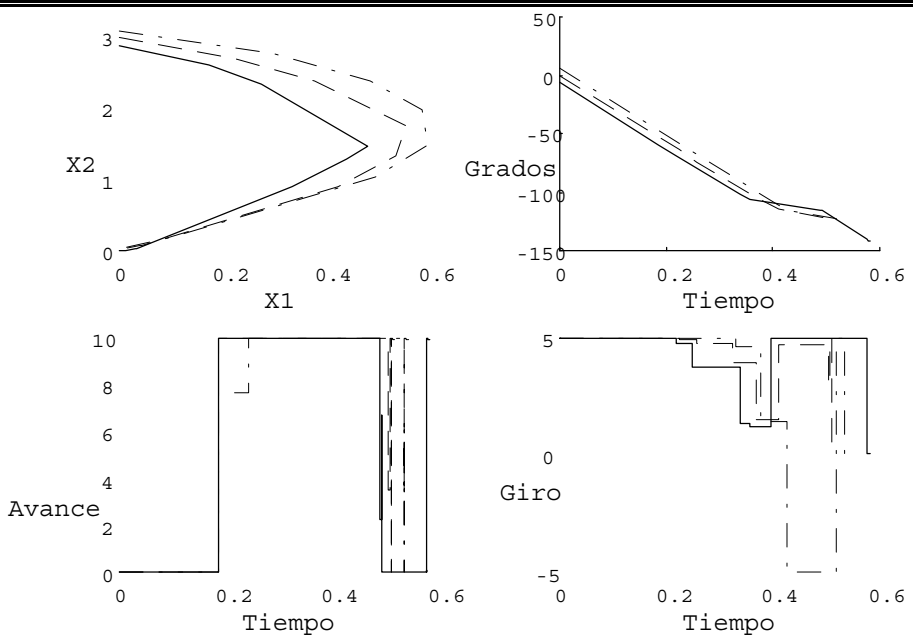

Figura 5.7 - Trayectorias para distintos puntos iniciales

permiten giros en ambos sentidos; y la neurona que produce el tiempo con sigmoide sin escala (rango [*0,1*]).

Con estas características el sistema se entrena rápidamente incluso utilizando el método de BP básico con momento fijo.

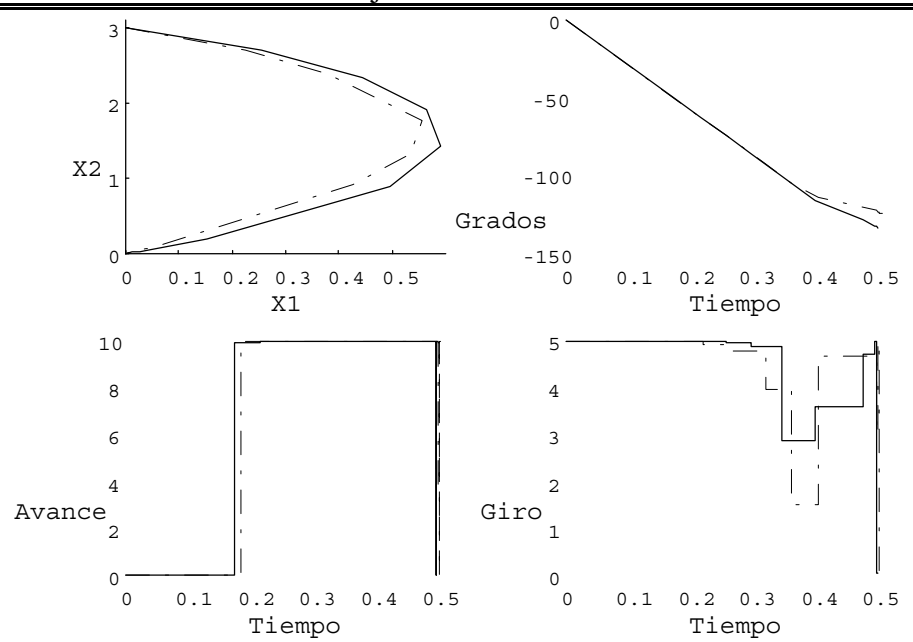

Figura 5.8 - Comparación de trayectorias cuando aumenta A^0.

En la figura 5.7 presentamos las trayectorias para distintos puntos iniciales. Como se observa, el sistema inicialmente gira sin avanzar. En una segunda fase continúa girando mientras avanza con el máximo comando. Al contrario de lo que ocurría con la PD, en

este caso, el sistema utiliza al máximo los comandos. Como consecuencia de ello emplea menos tiempo (entorno a *0.6* sg. frente a los *1.5* sg. de la PD) y por lo tanto se trata de trayectorias más óptimas.

Si se aumenta la región la trayectoria se degrada. En la figura 5.8 comparamos las trayectorias, desde el mismo punto inicial, para un sistema entrenado con distinto tamaño de región inicial. Se observa, para la generada en una región mayor, un aumento del tiempo empleado en llegar al punto final, implicando un aumento del costo.

Finalmente, en la figura 5.9 se muestran las trayectorias producidas cuando se introduce un punto exterior a la región entrenada comparadas con una trayectoria correcta. Como vemos, el sistema se desvía fuertemente de la trayectoria óptima y no alcanza el punto final deseado. Ocurre pues lo mismo que en el sistema anterior y se pone de manifiesto el problema de este método ante las perturbaciones.

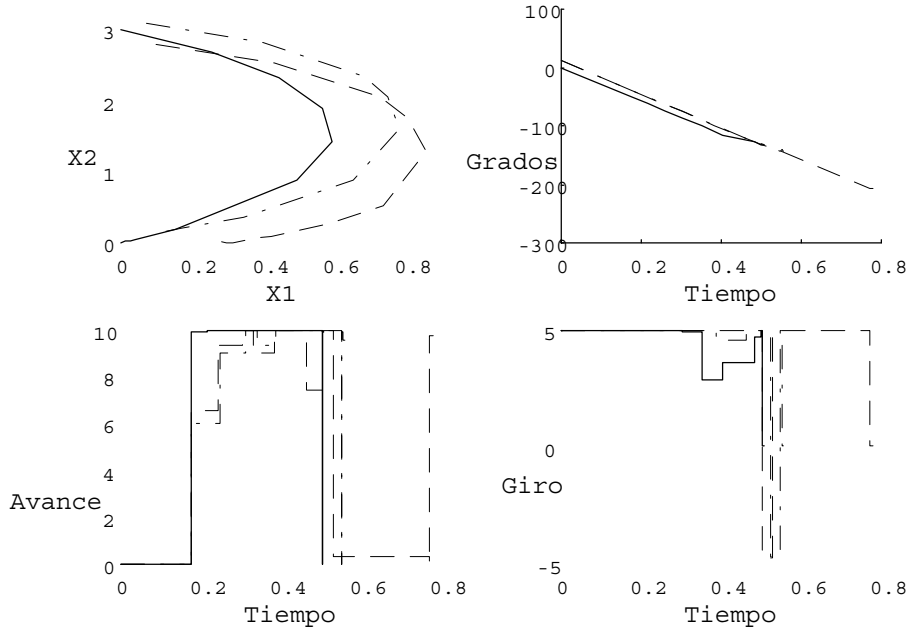

Figura 5.9 - Perturbación del punto inicial.

5.3.3 Tercer Sistema

5.3.3.1 Expresiones Necesarias

Las fórmulas discretizadas mediante diferencia adelante para este sistema, obtenidas en el capítulo 3, son las siguientes

$$\begin{cases} h_{1k+1} = h_{1k} + T_k\left[-R_1 P(\Delta h_k) + q_{1k}\right] \\ h_{2k+1} = h_{2k} + T_k\left[R_1 P(\Delta h_k) - R_2 P(h_{2k})\right] \\ \theta_{k+1} = \theta_k + T_k\left\{-\theta_k\left[1/R_T + C_p\left(R_1 P(\Delta h_k) - R_2 P(h_{2k})\right)\right] + q_{ck}\right\}/C_s \end{cases} \tag{5.48}$$

Para poder realizar la propagación del error a todas las redes debemos obtener las derivadas de la ecuación dinámica del sistema y de las funciones de costo con respecto a sus argumentos.

Las derivadas $\partial \underline{f} / \partial \underline{x}_k$ son las siguientes:

$$\frac{\partial h_{1k+1}}{\partial \underline{x}_k} = \begin{pmatrix} 1 - T_k R_1 P'(\Delta h_k) & T_k R_1 P'(\Delta h_k) & 0 \end{pmatrix} \tag{5.49}$$

$$\frac{\partial h_{2k+1}}{\partial \underline{x}_k} = \begin{pmatrix} T_k R_1 P'(\Delta h_k) & 1 - T_k\left[R_1 P'(\Delta h_k) + R_2 P'(h_{2k})\right] & 0 \end{pmatrix} \tag{5.50}$$

$$\frac{\partial \theta_{k+1}}{\partial h_{1k}} = -\frac{T_k}{C_s}\theta_k C_p R_1 P'(\Delta h_k) \tag{5.51}$$

$$\frac{\partial \theta_{k+1}}{\partial h_{2k}} = -\frac{T_k}{C_s} C_p \theta_k\left[R_1 P'(\Delta h_k)(-1) - R_2 P'(h_{2k})\right] \tag{5.52}$$

$$\frac{\partial \theta_{k+1}}{\partial \theta_k} = 1 + \frac{T_k}{C_s}\left[\frac{-1}{R_T} - C_p\left(R_1 P(\Delta h_k) - R_2 P(h_{2k})\right)\right] \tag{5.53}$$

Las derivadas $\partial \underline{f} / \partial \underline{v}_k$ son las siguientes:

$$\frac{\partial h_{1k+1}}{\partial \underline{v}_k} = \begin{pmatrix} T_k & 0 & -\left[R_1 P(\Delta h_k) - q_{1k}\right] \end{pmatrix} \tag{5.54}$$

$$\frac{\partial h_{2k+1}}{\partial \underline{v}_k} = \begin{pmatrix} 0 & 0 & R_1 P(\Delta h_k) - R_2 P(h_{2k}) \end{pmatrix} \tag{5.55}$$

$$\frac{\partial \theta_{k+1}}{\partial \underline{v}_k} = \begin{pmatrix} 0 & \frac{T_k}{C_s} & \frac{1}{C_s}\left\{\theta_k\left[-1/R_T - C_p\left(R_1 P(\Delta h_k) - R_2 P(h_{2k})\right)\right] + q_{ck}\right\} \end{pmatrix} \tag{5.56}$$

Exigimos trayectorias de tiempo mínimo también a este sistema ($L(\cdot)=T_k$). Sus derivadas serán

$$\frac{\partial L}{\partial \underline{x}_k} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \tag{5.57}$$

$$\frac{\partial L}{\partial \underline{v}_k} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{5.58}$$

Como comentamos al tratar este problema mediante PD, el fijar un caudal y temperatura deseada es equivalente a fijar las variables h_2 y θ; y por exigir un punto final estable se fija h_1. Si denominamos h_1^f, h_2^f y θ^f a dichos valores finales, la función de costo para el punto N será

$$S(\underline{x}_N, N) = V_N \cdot \left[(h_1^f - h_{1N})^2 + (h_2^f - h_{2N})^2 + (\theta^f - \theta_N)^2 \right] \tag{5.59}$$

Sus derivadas son

$$\frac{\partial S}{\partial \underline{x}_N} = \begin{pmatrix} 2V_N h_{1N} & 2V_N h_{2N} & 2V_N \theta_N \end{pmatrix} \tag{5.60}$$

5.3.3.2 Resultados

Fijamos los parámetros a los mismos valores que utilizamos para la PD. Elegimos un valor N=15 con redes iguales de una capa oculta de 5 neuronas y una capa de salida de 3, igual al número de comandos. Tanto los comandos como el tiempo sólo pueden tomar valores positivos por lo que todas las funciones de activación de la capa de salida son sigmoides. La escala usada para los caudales es de *50* mientras que para el tiempo es de sólo *10*.

Para el punto final fijamos unos valores de h_2^f=*5* y θ^f=*5*; por este valor de h_2^f y por los de R_1 y R_2 tenemos que ha de ser h_1^f=*5*. Las regiones iniciales las tomaremos, de distinto tamaño, en torno al origen.

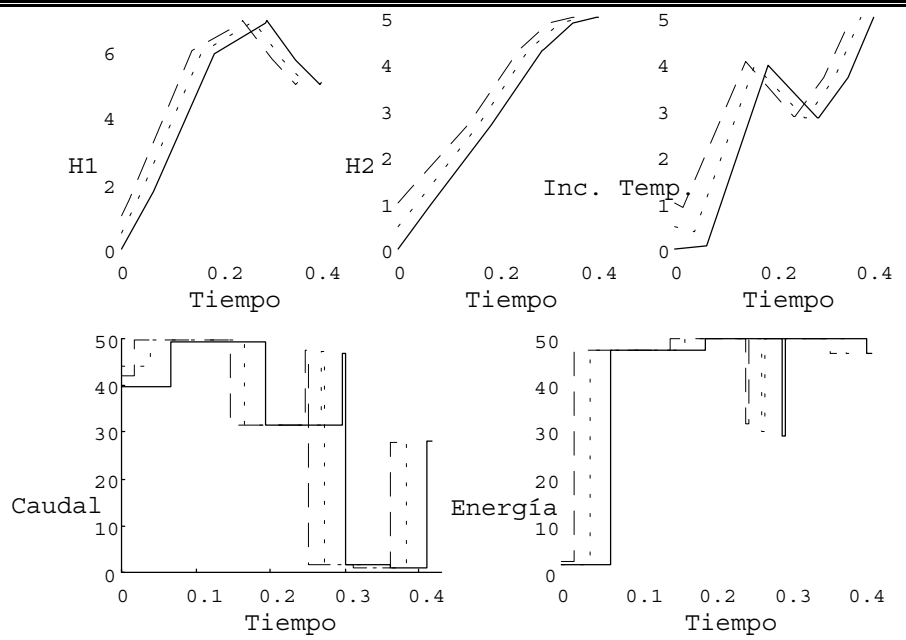

Figura 5.10 - Trayectorias para distintos puntos iniciales

Para el entrenamiento utilizamos la BP con un gradiente conjugado con factor tipo Fletcher-Reeves con reseteo por proximidad de *0.01*. Esto quiere decir que inicializamos la dirección de búsqueda cuando dos gradientes sucesivos no difieren en más de dicha cantidad. También, a lo largo del entrenamiento, reducimos el factor η de forma manual, a la vista del avance del sistema en una tanda de entrenamientos. De esta manera, si tras un conjunto importante de entrenamientos (cien mil aproximadamente), el costo de las trayectorias no descendía significativamente reducíamos el valor de η. Llegamos a variar desde valores iniciales de la unidad hasta valores mínimos de 5×10^{-3}

En la figura 5.10 se muestran las trayectorias obtenidas, con estas condiciones, para distintos puntos iniciales de la región inicial A^0. La forma de la trayectoria es similar a la obtenida mediante PD (apartado 4.2.3.4) salvo que los comandos ahora son más enérgicos. Esto se observa sobre todo en el suministro de energía al segundo tanque que, en este caso, sí alcanza el valor máximo permitido. Además, la aplicación de este comando comienza antes. En cuanto al caudal de líquido también hay diferencia, ya que la aplicación del valor máximo se prolonga en este caso. Como consecuencia de todo esto las trayectorias tardan alrededor de *0.4* sg. frente a los cerca de *0.8* sg. que tardan las de PD, teniendo por ello menos coste.

En la figura 5.11 se comparan las trayectorias para el punto de partida *(0,0,0)* cuando la cadena de redes ha sido entrenada con una región inicial mayor. Se observa que, en el caso de región mayor, no se llega a aplicar todo el caudal permitido, prolongándose, por ello, su duración.

Finalmente, en la figura 5.12 se compara la trayectoria generada para un punto dentro

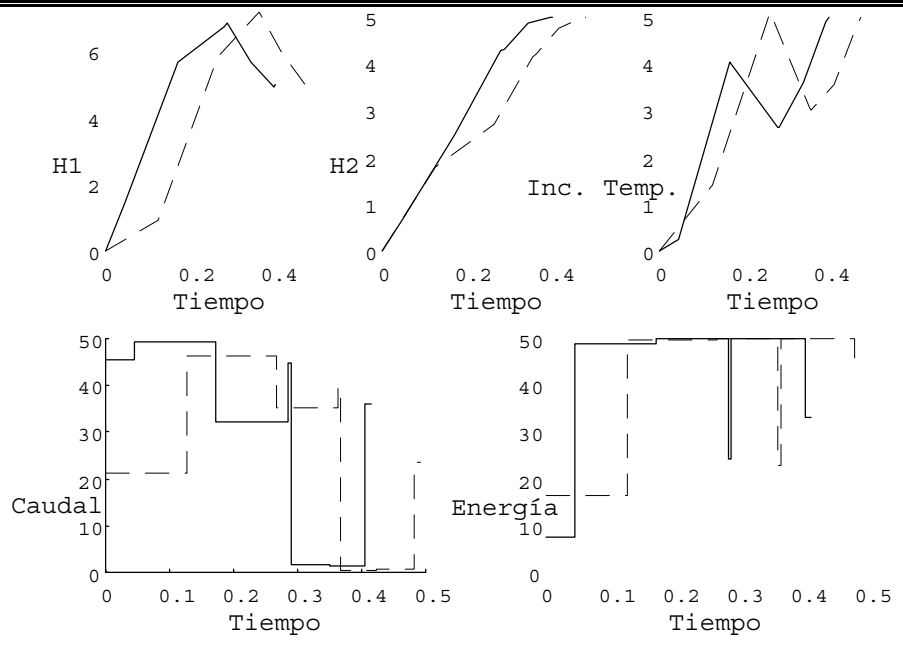

Figura 5.11 - Comparación de trayectorias al aumentar A^0.

de la región inicial con otras para puntos fuera de ella. Se observa claramente que la forma de la trayectoria correcta se reproduce en las otras, y por ello no se llega al punto final deseado en ninguna de las trayectorias perturbadas.

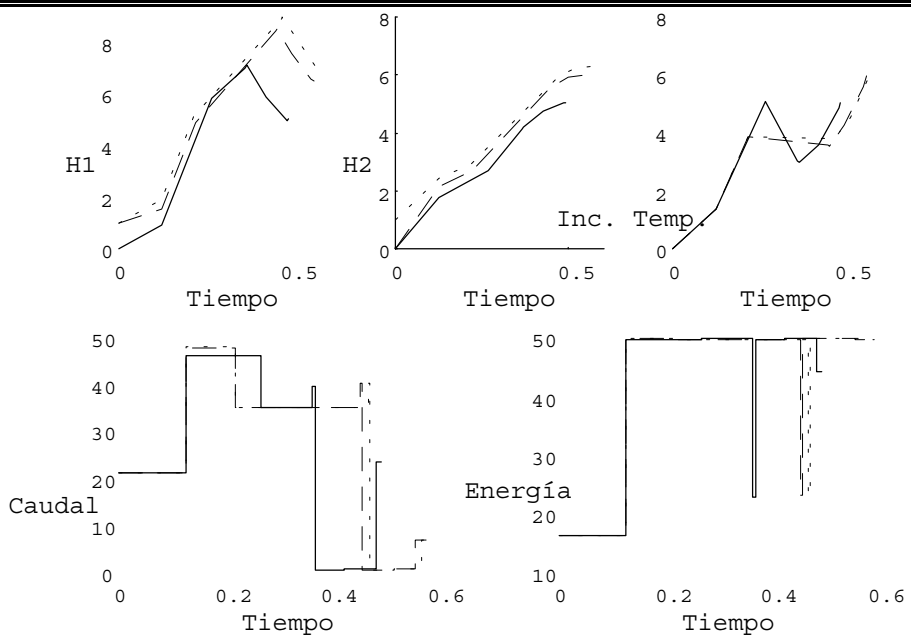

Figura 5.12 - Perturbación del punto inicial.

5.4 Resumen y Conclusiones

De la aplicación de este primer método a los problemas ejemplo podemos concluir las siguientes características:

- El entrenamiento de la cadena de RN es rápido y sencillo, sin la necesidad de utilizar algoritmos complejos de la BP. Este entrenamiento se complica ligeramente cuando aumenta la complejidad del sistema.
- Los resultados obtenidos en todos los sistemas son del mismo o mejor costo que en el caso de la PD. Esta mejoría aumenta con la complejidad del sistema. El motivo de esto es, principalmente, la limitación de memoria en el caso de la PD, que no le permite una discretización suficiente en sistemas de tercer orden. En cambio, en este método, la cadena de redes, con mucha menos memoria, se adapta a la región que cubren las trayectorias óptimas para los puntos de su región inicial consiguiendo reducir los costes.
- El método tiene un serio problema cuando se presenta un punto que no pertenece a la región entrenada. En ese caso se reproduce la forma de las trayectorias aprendidas y por lo tanto el punto final queda muy lejos del deseado.

Se puede decir, entonces, que este método sacrifica la generalidad que posee la PD, que debe cubrir amplias regiones del espacio de estados, en favor de unos menores requerimientos de memoria. Con ello obtiene resultados más óptimos, ya que se ciñe a los puntos considerados, aunque produce malos resultados fuera de ellos.

La forma de poder aplicar este método es elegir una región inicial lo suficientemente grande para garantizar que las perturbaciones presentes en el sistema no lo saquen de las regiones consideradas. Esto implica que la optimalidad disminuirá al aumentar la magnitud de las perturbaciones. Esta relación determina la adecuación de este método en cada caso particular.

6

Segundo Método Basado en Redes Neuronales

El método presentado en el capítulo anterior tiene dos inconvenientes. El primero es la imposibilidad de utilizar la formulación BPF para discretizar las ecuaciones del sistema. No podemos beneficiarnos de las propiedades demostradas por esta discretización en la PD. Un objetivo es diseñar un método similar al anterior que si nos permita su utilización.

El segundo, y principal, inconveniente del método anterior es la escasa robustez de la solución cuando el sistema se presenta en un estado que no pertenece a ninguna de las regiones A^i cubiertas por las distintas etapas. Esto hace que su aplicación en sistemas reales se presente como poco útil.

6.1 Descripción del Método

Como hemos dicho una de las motivaciones en el diseño de este segundo método es poder utilizar la formulación BPF para discretizar las ecuaciones del sistema. Esta formulación conseguía obtener expresiones para el comando y el tiempo en función del estado actual y el siguiente. En la PD esto evitaba la necesidad de interpolar en el

primer barrido de la PD. Además había demostrado unos mejores resultados que con la discretización clásica.

Utilizando la BPF las ecuaciones dinámicas se pueden poner de la forma

$$\underline{v}_{k+1} = (\underline{u}_k, T_k) = \underline{f}^{-1}(\underline{x}_k, \underline{x}_{k+1}, \underline{v}_{k-1}, k) \tag{6.1}$$

en donde se calculan el comando e intervalo de tiempo necesario para realizar la transición del estado $\underline{x}_k$ al $\underline{x}_{k+1}$, teniendo también en cuenta el comando de la etapa anterior. Si queremos utilizar estas expresiones en una cadena de redes neuronales, similares a las del método anterior, ahora las redes han de producir el estado siguiente a partir del actual y son las ecuaciones dinámicas las que se encargan de obtener el comando y tiempo necesario. Es decir ha de ser ahora

$$\underline{x}_{i+1} = \gamma(\underline{w}_i, \underline{x}_i) \tag{6.2}$$

para la etapa i-ésima. El esquema completo, si tenemos N etapas, es el presentado en la figura 6.1.

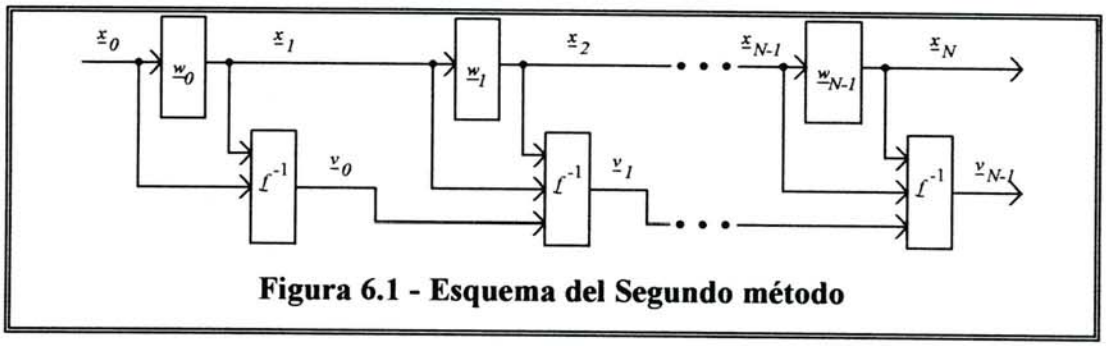

Figura 6.1 - Esquema del Segundo método

De esta manera lo que aprende la cadena de redes es la trayectoria óptima para el conjunto de estados de una región inicial $A^0 \subset \mathbb{R}^n$. Los estados de las sucesivas etapas de estas trayectorias óptimas pertenecerán a las regiones A^i, con $i=1,\ldots,N$.

El tipo de valores y minimización que deben producir las redes es similar al del método anterior, por lo que optamos también por perceptrones multicapa. La ecuación de su neurona era

$$z_q^i(s) = \underline{w}_q^i(s)^{\mathrm{T}}\, \underline{y}^i(s-1) + w_{0q}^i(s); \quad y_q^i(s) = g(z_q^i(s)) \tag{6.3}$$

En este caso, eligiendo el tipo de función de activación en las neuronas de salida, podemos determinar el rango de variación de las variables de estado. Por este método se pueden implementar restricciones en el espacio de estados. Las restricciones en los comandos e intervalos de tiempo, en cambio, han de fijarse en la función de costo. Esto hará necesario unas funciones de costo mucho más complejas. El punto final también será fijado por medio de las función de costo de la última etapa. De cualquier manera, como en el método anterior, las fórmulas aquí presentadas serán válidas para el caso más general, independiente de la expresión de las distintas funciones.

6.1.1 Cálculo de las Trayectorias

El proceso de cálculo es similar al del método anterior, con la diferencia de que nos basta con la cadena de redes para conocer la trayectoria para un estado inicial dado. La ecuación dinámica del sistema la utilizaremos únicamente para calcular el comando e intervalo de tiempo necesarios para cada una de las transiciones de estados. El proceso de cálculo será:

Elegir el estado inicial $\underline{x}_0$
Para i=0 **hasta** N-1
  Tomar $\underline{y}_i(0)=\underline{x}_i$
  Para s=1 **hasta** L^i
    aplicar (6.3) para obtener $\underline{y}_i(s)$
  Tomar $\underline{x}_{i+1}=\underline{y}_i(L^i)$
  Obtener $\underline{v}_i$ aplicando (6.1)

De esta manera obtenemos la trayectoria $\{\underline{x}_0,\underline{x}_1,\ldots,\underline{x}_{N-1}\}$ y la secuencia de comandos e intervalos de tiempo aplicados $\{\underline{v}_0,\underline{v}_1,\ldots,\underline{v}_{N-1}\}$.

El hecho de que la salida de una red sea directamente la entrada de la siguiente hace posible ver esta cadena como una única red de la cual se toman resultados de capas intermedias. Esta visión no aporta ninguna ventaja, ni al proceso de cálculo ni al de entrenamiento, por lo que optamos por tratarlas como redes individuales en ambos procesos.

6.1.2 Proceso de Entrenamiento

Para el proceso de entrenamiento nos encontramos en la misma situación que en el método anterior. Los vectores de entrenamiento son los puntos de la región inicial A^0. Para estos vectores de entrada no conocemos las vectores de salida deseada sino que poseemos directamente la función de costo

$$J = S(\underline{x}_N, N) + \sum_{i=0}^{N-1} L(\underline{x}_i, \underline{v}_i, i) \tag{6.4}$$

Esta función hace las veces de función de error de los métodos de entrenamiento supervisado.

También en este caso utilizaremos un entrenamiento tipo BP. Vamos, a continuación, a deducir las fórmulas para obtener el gradiente del costo con respecto a cada peso

$$\frac{\partial J}{\partial w_{pq}^i(s)} = \delta_q^i(s) y_p^i(s-1) \tag{6.5}$$

La expresión de δ para las capas ocultas de cada red ($s=1,\ldots,L^i$) es como en el caso general

$$\delta_q^i(s)=\frac{\partial J}{\partial z_q^i(s)}=\frac{\partial J}{\partial y_q^i(s)}\frac{\partial y_q^i(s)}{\partial z_q^i(s)}=\left[\sum_{h=1}^{n_{s+1}^i}\delta_h^i(s+1)w_{qh}^i(s+1)\right]g'(z_q^i(s)) \tag{6.6}$$

Para la capa de salida en esta ocasión $y_q^i(L^i)=[\underline{x}_{i+1}]_q$ con lo que podemos poner que

$$\delta_q^i(L^i)=\frac{\partial J}{\partial z_q^i(L^i)}=\frac{\partial J}{\partial y_q^i(L^i)}\frac{\partial y_q^i(L^i)}{\partial z_q^i(L^i)}=\left[\frac{\partial J}{\partial \underline{x}_{i+1}}\right]_q g'(z_q^i(L^i)) \tag{6.7}$$

Si definimos, de la misma manera que en el método anterior, la variable $\underline{\lambda}_i$ como

$$\underline{\lambda}_i\equiv\nabla_{\underline{x}_i}J=\left(\frac{\partial J}{\partial \underline{x}_i}\right)^{\mathrm{T}} \tag{6.8}$$

podremos poner que

$$\delta_q^i(L^i)=\left[\underline{\lambda}_{i+1}\right]_q g'(z_q^i(L^i)) \tag{6.9}$$

Debemos obtener una expresión para $\underline{\lambda}_i$. Esto lo hacemos aplicando la regla de la cadena. Basándonos en la figura 6.2, donde se añade al esquema el cálculo de la función de costo, podemos poner que

$$\underline{\lambda}_i=\frac{\partial J}{\partial \underline{x}_i}=\frac{\partial J}{\partial \underline{y}^i(0)}\frac{\partial \underline{y}^i(0)}{\partial \underline{x}_i}+\frac{\partial L(\underline{x}_i,\underline{v}_i)}{\partial \underline{x}_i}+\frac{\partial J}{\partial \underline{v}_{i-1}}\frac{\partial \underline{v}_{i-1}}{\partial \underline{x}_i}+\frac{\partial J}{\partial \underline{v}_i}\frac{\partial \underline{v}_i}{\partial \underline{x}_i} \tag{6.10}$$

valido para $i=1,\ldots,N-1$. En esta expresión el primer sumando es la aportación por ser entrada a la red i-ésima; el segundo es debido al costo que genera $\underline{x}_i$ en esa etapa; el tercero y cuarto son la aportación por generar $\underline{v}_{i-1}$ y $\underline{v}_i$ respectivamente. Si definimos el nuevo vector $\underline{\omega}_i$ como

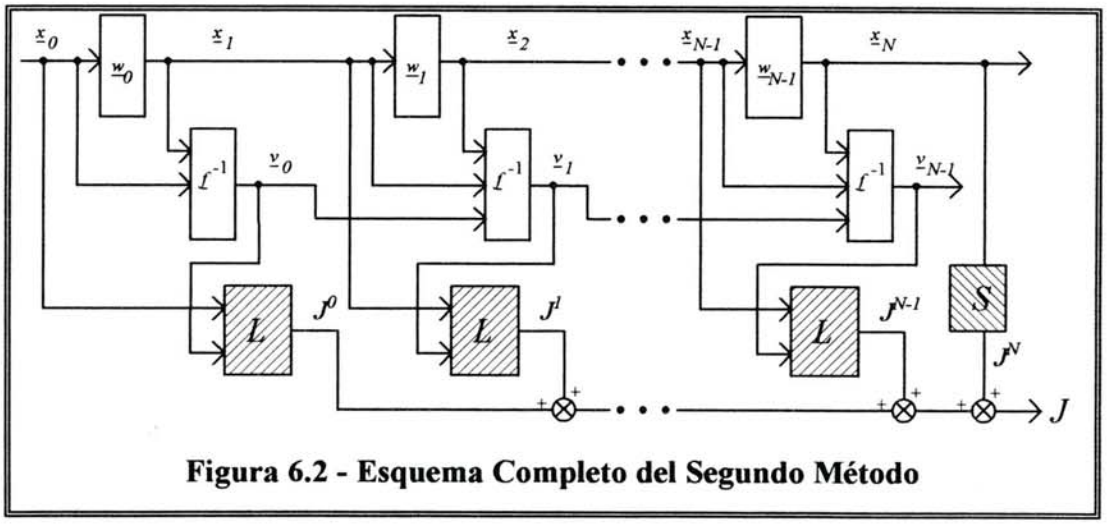

Figura 6.2 - Esquema Completo del Segundo Método

$$\underline{\omega}_i \equiv \nabla_{\underline{v}_i} J = \left(\frac{\partial J}{\partial \underline{v}_i} \right)^{\mathrm{T}} \tag{6.11}$$

la expresión anterior quedará

$$\underline{\lambda}_i = \frac{\partial J}{\partial \underline{y}^i(0)} \frac{\partial \underline{y}^i(0)}{\partial \underline{x}_i} + \frac{\partial L(\underline{x}_i, \underline{v}_i)}{\partial \underline{x}_i} + \underline{\omega}_{i-1}^{\mathrm{T}} \frac{\partial \underline{v}_{i-1}}{\partial \underline{x}_i} + \underline{\omega}_i^{\mathrm{T}} \frac{\partial \underline{v}_i}{\partial \underline{x}_i} \tag{6.12}$$

Para el primer sumando debemos tener en cuenta que $\underline{y}_i(0)=\underline{x}_i$ y aplicar la regla de la cadena dentro de las neuronas de la primera capa

$$\frac{\partial J}{\partial \underline{y}^i(0)} \frac{\partial \underline{y}^i(0)}{\partial \underline{x}_i} = \left[\sum_{q=1}^{n} \frac{\partial J}{\partial z_q^i(1)} \frac{\partial z_q^i(1)}{\partial \underline{y}^i(0)} \right] \cdot 1 = \sum_{q=1}^{n} \delta_q^i(1) \underline{w}_q^i(1)^{\mathrm{T}} \tag{6.13}$$

Para el tercer y cuarto sumandos, teniendo en cuenta (6.1), se podrán poner como

$$\underline{\omega}_{i-1}^{\mathrm{T}} \frac{\partial \underline{v}_{i-1}}{\partial \underline{x}_i} = \underline{\omega}_{i-1}^{\mathrm{T}} \frac{\partial \underline{f}^{-1}(\underline{x}_{i-1}, \underline{x}_i, \underline{v}_{i-2}, i-1)}{\partial \underline{x}_i} \tag{6.14}$$

$$\underline{\omega}_i^{\mathrm{T}} \frac{\partial \underline{v}_i}{\partial \underline{x}_i} = \underline{\omega}_i^{\mathrm{T}} \frac{\partial \underline{f}^{-1}(\underline{x}_i, \underline{x}_{i+1}, \underline{v}_{i-1}, i)}{\partial \underline{x}_i} \tag{6.15}$$

respectivamente. La expresión completa para $\underline{\lambda}_i$ queda finalmente

$$\underline{\lambda}_i = \sum_{q=1}^{n} \delta_q^i(1) \underline{w}_q^i(1)^{\mathrm{T}} + \frac{\partial L(\underline{x}_i, \underline{v}_i)}{\partial \underline{x}_i} + \underline{\omega}_{i-1}^{\mathrm{T}} \frac{\partial \underline{f}^{-1}(\underline{x}_{i-1}, \underline{x}_i, \underline{v}_{i-2}, i-1)}{\partial \underline{x}_i} + \underline{\omega}_i^{\mathrm{T}} \frac{\partial \underline{f}^{-1}(\underline{x}_i, \underline{x}_{i+1}, \underline{v}_{i-1}, i)}{\partial \underline{x}_i} \tag{6.16}$$

La expresión de $\underline{\lambda}$ para $i=N$ queda directamente como la parcial de la función de costo final

$$\underline{\lambda}_N = \frac{\partial J}{\partial \underline{x}_N} = \frac{\partial S(\underline{x}_N, N)}{\partial \underline{x}_N} \tag{6.17}$$

Falta determinar ahora la expresión para la nueva variable $\underline{\omega}_i$. Tomando la definición y aplicando nuevamente la regla de la cadena, a la vista de la figura 6.2, tendremos que

$$\underline{\omega}_i^{\mathrm{T}} = \frac{\partial J}{\partial \underline{v}_i} = \frac{\partial L(\underline{x}_i, \underline{v}_i)}{\partial \underline{v}_i} + \frac{\partial J}{\partial \underline{v}_{i+1}} \frac{\partial \underline{v}_{i+1}}{\partial \underline{v}_i} \tag{6.18}$$

válida para $i=0,\ldots,N-2$. El segundo sumando, que representa la aportación por generar el comando $\underline{v}_{i+1}$, se puede poner

$$\frac{\partial J}{\partial \underline{v}_{i+1}} \frac{\partial \underline{v}_{i+1}}{\partial \underline{v}_i} = \underline{\omega}_{i+1}^{\mathrm{T}} \frac{\partial \underline{f}(\underline{x}_{i-1}, \underline{x}_i, \underline{v}_i, i-1)}{\partial \underline{v}_i} \tag{6.19}$$

Llevando este resultado a la ecuación anterior tendremos que:

$$\underline{\omega}_i^{\mathrm{T}} = \frac{\partial L(\underline{x}_i, \underline{v}_i)}{\partial \underline{v}_i} + \underline{\omega}_{i+1}^{\mathrm{T}} \frac{\partial \underline{f}(\underline{x}_{i-1}, \underline{x}_i, \underline{v}_i, i-1)}{\partial \underline{v}_i} \tag{6.20}$$

Esta expresión será válida también para $i=N-1$ si hacemos $\underline{\omega}_N=0$. Con ello tenemos una recurrencia desde $N-1$ hasta 0.

El proceso completo de un paso de entrenamiento en este método será el siguiente:

Elegir un estado inicial $\underline{x}_0 \in A^0$
Calcular la trayectoria para ese estado inicial
Hacer $\underline{\omega}_N=0$
Para i=N-1 **hasta** 0
    Calcular $\underline{\omega}_i$ mediante (6.20)
Determinar $\underline{\lambda}_N$ mediante (6.17)
Para i=N-1 **hasta** 0
    Calcular δ de la última capa mediante (6.9)
    Para s=L^i-1 **hasta** 1
        Calcular δ de la capa s mediante (6.6)
    Calcular $\underline{\lambda}_i$ mediante (6.16)
Determinar parcial del costo con respecto a cada peso mediante (6.5)
Aplicar la corrección de los pesos según el algoritmo elegido

A diferencia del método primero, en este caso hemos tenido que definir una nueva variable $\underline{\omega}$, asociada a cada etapa y relacionada con los comandos, y que produce una expresión recursiva. Esta variable no depende de la propagación del error dentro de las redes sino que puede obtenerse independientemente. Por otro lado la expresión de $\underline{\lambda}$ ahora no es recursiva sino que se expresa en función de $\underline{\omega}$.

Como en el método anterior, es necesario conocer las derivadas parciales de todos las funciones con respecto a todos sus argumentos. En este caso la función de la dinámica inversa $\underline{f}^1$ posee tres argumentos vectoriales. Esto, junto a la utilización de más variables, hace que este método sea mucho mas laborioso en el cálculo.

Siguiendo un razonamiento similar al del apartado 5.2.4, vemos que aumentar la dimensión del sistema significa aumentar el número de entradas y salidas de la red. Estas nuevas neuronas incrementan el número de pesos en un valor constante que se añade al que ya tenemos. Como la complejidad viene dada por el número de pesos, ésta crece linealmente con la dimensión del sistema. Por esta razón este método es muy ventajoso con respecto a la PD ya que, como vimos, ésta posee complejidad exponencial.

6.1.3 Modificación de la Función de Costo

Al generar las redes los estados, eligiendo sus funciones de activación, lo que podemos limitar es el rango de las variables de estado. Al contrario de lo que ocurría en el método anterior, los comandos e intervalos de tiempo se han de limitar en la función de costo para que los no permitidos sean descartados en el proceso de entrenamiento.

Cuando las redes están lejos de la solución, al principio del entrenamiento, la transición que propone una etapa, es decir, la relación del estado a la salida con el estado a la entrada, puede demandar del sistema valores del comando e intervalo de tiempo fuera de los permitidos. El caso más notable, por lo claramente inadmisible, es la necesidad de intervalos de tiempo negativos. Esta situación ha de ser gravada por encima de cualquier otra en todas las etapas.

Si denominamos L^d a la función de coste que se desea para el sistema, la función de costo real J se debe definir como

$$J = S(\underline{x}_N, N) + \sum_{i=0}^{N-1} \left[L^d + L^T + L^u \right] \tag{6.21}$$

donde L^T representa una función que pesa los tiempos negativos y L^u una función que grava los comandos no admisibles. La expresión de L^u será dependiente de la forma que tienen las restricciones de los comandos y por lo tanto ha de definirse a la vista de cada problema.

Por otro lado, L^T debe aparecer en todos los casos y con una expresión similar. Una primera elección de la función L^T sería tomarla de tipo potencial con los tiempos negativos, es decir

$$L^T(\underline{x}_i, \underline{v}_i, i) = L^T(T_i) = \begin{cases} 0 & si \quad T_i \geq 0 \\ V_T(-T_i)^P & si \quad T_i < 0 \end{cases} \tag{6.22}$$

donde $P \in \mathbf{N}$ y $P>1$. Parece lógico elegir valores de P grandes, lo que significaría grandes costos, pero esto presenta dos dificultades. Es su derivada la que será utilizada en el proceso de entrenamiento. Ésta tiene la expresión

$$\frac{\partial L^T(\underline{x}_i, \underline{v}_i, i)}{\partial T_i} = \frac{dL^T(T_i)}{dT_i} = \begin{cases} 0 & si \quad T_i \geq 0 \\ -PV_T(-T_i)^{P-1} & si \quad T_i < 0 \end{cases} \tag{6.23}$$

Al comienzo del entrenamiento el valor de T_i puede ser un valor muy negativo por lo tanto el valor de su derivada se puede llegar a hacer excesivamente grande. Esto supone modificaciones muy fuertes en los pesos que llevan fácilmente a las neuronas a un estado de saturación. En el otro extremo, en cambio, para valores de T_i menores que 1 la corrección será menor cuanto mayor es P, ya que la multiplicación de dos números

menores que 1 da un número menor que cualquiera de los dos multiplicados. Esta disminución se hace mayor cuanto más cerca del 0 estemos. La situación que se producirá entonces es que, si las redes no se saturan, los valores de T_i negativos pero pequeños, al avanzar hacia su eliminación costará cada vez más eliminarlos. Por muy grande que elijamos la constante V_T, se llegará a la situación en que los otros sumandos del coste tienen mayor importancia en el entrenamiento que aquél que pesa el tiempo negativo.

A la vista de este comportamiento optamos por elegir una función lineal para L^T, es decir

$$L^T(\underline{x}_i, \underline{v}_i, i) = L^T(T_i) = \begin{cases} 0 & si \quad T_i \geq 0 \\ V_T(-T_i) & si \quad T_i < 0 \end{cases} \tag{6.24}$$

equivalente a hacer $P=0$ en la fórmula anterior. Esto representa una recta de pendiente V_T que pasa por el origen. La expresión de su derivada es

$$\frac{\partial L^T(\underline{x}_i, \underline{v}_i, i)}{\partial T_i} = \frac{dL^T(T_i)}{dT_i} = \begin{cases} 0 & si \quad T_i \geq 0 \\ -V_T & si \quad T_i < 0 \end{cases} \tag{6.25}$$

En este caso, como vemos, la corrección es constante, independientemente del valor de T_i, lo que asegura una reducción paulatina e indefectible de los tiempos negativos. En la figura 6.3 se comparan esta función y su derivada con las propuestas anteriormente para distintos valores de P. Como vemos aunque el costo con las primeras supera rápidamente al de la función lineal, en la zona de T_i>-1 la diferencia está claramente a favor de la función lineal.

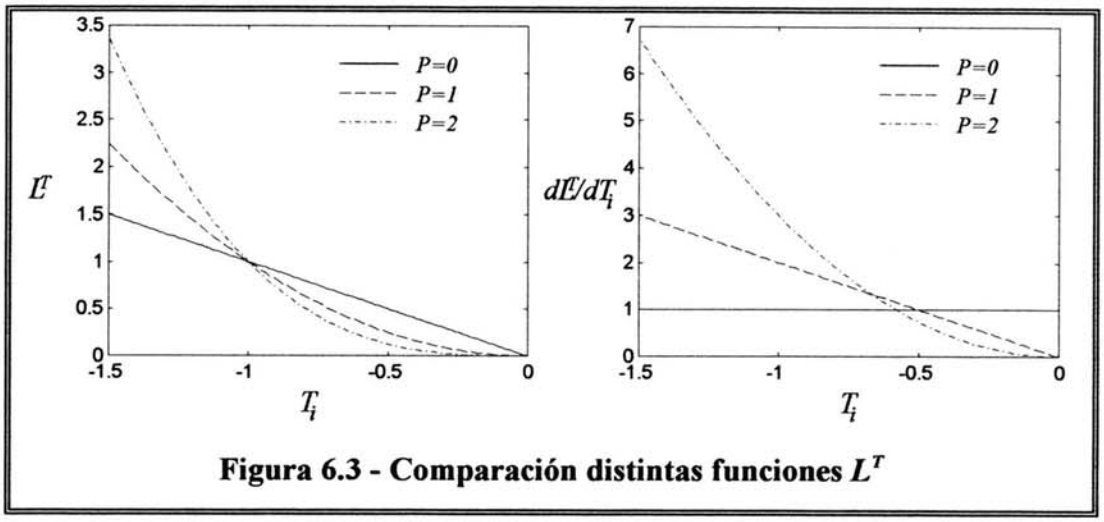

Figura 6.3 - Comparación distintas funciones L^T

A efectos prácticos, para reconocer claramente las situaciones en que aún existen tiempos negativos añadimos una constante grande a L^T para los valores negativos de T_i, es decir, tomamos

$$L^T(\underline{x}_i,\underline{v}_i,i)=L^T(T_i)=\begin{cases}0 & si \quad T_i\geq 0\\ D_T+V_T(-T_i) & si \quad T_i<0\end{cases} \tag{6.26}$$

Esta constante no modifica la expresión de la derivada pero, mientras existan tiempos negativos en alguna de las etapas, el coste total será del orden de D_T. Aunque esta nueva definición da lugar a una función no continua, como nosotros indicamos la forma de las distintas derivadas al programa, si elegimos la definición (6.25) la derivada estará definida para todos los puntos, incluido el 0. De esta manera el entrenamiento se produce como si se tratase de una función continua y derivable.

Por lo dicho para L^T, es conveniente plantear una expresión similar para las funciones L^u. Si el comando i-ésimo ha de estar limitado al rango $[U^i_{min}, U^i_{max}]$ haremos

$$L^u(\underline{x}_i,\underline{v}_i,i)=L^u(u_i)=\begin{cases}V_i(u_i-U^i_{max}) & si \quad u_i\geq U^i_{max}\\ 0 & si \quad U^i_{max}>u_i>U^i_{min}\\ V_i(U^i_{min}-u_i) & si \quad u_i\leq U^i_{max}\end{cases} \tag{6.27}$$

y usaremos una expresión para su derivada de manera que esté definida en todos sus puntos

$$\frac{\partial L^u(\underline{x}_i,\underline{v}_i,i)}{\partial u_i}=\begin{cases}V_i & si \quad u_i\geq U^i_{max}\\ 0 & si \quad U^i_{max}>u_i>U^i_{min}\\ -V_i & si \quad u_i\leq U^i_{max}\end{cases} \tag{6.28}$$

Al igual que antes, se garantiza que existirá una corrección suficiente hasta que el comando esté en la zona permitida.

6.1.4 Problemas del Entrenamiento

Por la estructura de este método, el mayor número de variables intermedias a manejar, el mayor número de derivadas necesarias y los nuevos sumandos añadidos a la función de costo, el proceso de entrenamiento se hace largo y difícil. Cuando tratamos con sistemas complejos resulta prácticamente imposible llegar a soluciones óptimas a partir de pesos iniciales aleatorios. Incluso en sistemas sencillos es preciso aplicar alguna de las técnicas que veremos más adelante para que el sistema converja en un número de entrenamientos razonable. Estudiando la falta de evolución en el entrenamiento descubrimos que ésta se debe a dos causas principales que veremos en los siguientes apartados.

6.1.4.1 Estancamiento en Mínimos Locales

Aunque éste es un problema común al entrenamiento con BP, y en general a cualquier método descendente, en este caso se agrava. Debido a la forma de las derivadas la superficie de error posee una rugosidad inimaginable.

Para poder hacernos una idea estudiamos el comportamiento de la función de costo sólo en una dirección. Cuando elegimos la dirección del gradiente para hacer el estudio, en la que se supone que la variación de la superficie es la mayor, observamos que este comportamiento era prácticamente infinitesimal. Sólo para valores unos órdenes de magnitud por encima de la resolución que nos da la doble precisión la función de costo bajaba. A partir de allí surgía un nuevo máximo. Cuando intentamos aplicar las técnicas de búsqueda en la dirección del gradiente el paso de avance se reducía a valores infinitesimales que producían un avance nulo. Si miramos en un orden superior, se observa que la dirección elegida para el gradiente no producía una reducción del coste apreciable.

La explicación para este comportamiento viene dada por la forma de las propias derivadas. El comportamiento de éstas, incluso variando únicamente uno de sus argumentos, varía muy rápidamente. Al combinar la expresión de éstas para formar la superficie de error, 500-dimensional aproximadamente, hace que sea extremadamente rugosa y llena de mínimos locales infinitesimales.

Una forma de paliar este problema, en cierta medida, es promediar varios gradientes antes de aplicar la corrección a los pesos. Se presentan varias decenas de puntos de la región inicial A^0 a la cadena y se calcula el gradiente correspondiente pero sin aplicar la corrección a los pesos. De ese conjunto de gradientes se hace el promedio obteniéndose así la dirección que, una vez filtradas las variaciones de menor orden, hace decrecer el costo macroscópicamente.

También es preciso prescindir de los métodos de búsqueda en la dirección del gradiente que detendrían el avance indefectiblemente. El control del paso de avance ha de hacerse mediante prueba y error hasta alcanzar valores satisfactorios. Estos valores pueden diferir notablemente entre distintas fases del entrenamiento.

La alternativa lógica para evitar los mínimos locales, como es el algoritmo de enfriamiento progresivo, debido a la presencia de gran número de mínimos locales hace necesario un ritmo de reducción de la temperatura artificial lento. Esto, junto con el gran número de pesos, da lugar a necesidades de cálculo prohibitivas.

6.1.4.2 Falta de Sensibilidad

Una situación que puede surgir con largos periodos de entrenamiento es la pérdida de sensibilidad de las redes. Al no existir variación importante en los valores de la entrada y la salida, por el escaso aprendizaje, cada red se limita a modificarse de manera que la salida de el valor esperado sin tener en cuenta la entrada. A esta situación se llega porque el peso de sesgo de las neuronas de alguna de las capas se hace grande con respecto a los restantes pesos de esa neurona. De esta manera la importancia de las neuronas de la capa anterior se hace mínima y el valor constante del sesgo determina la salida.

Una forma de detectar esta anomalía consiste en hacer pruebas de sensibilidad por capas cada ciertos entrenamientos. Esto conlleva el calcular cada capa para unas entradas variadas. La sensibilidad será el promedio de variación de la salida con la variación a la entrada.

Una forma más sencilla para detectar esta situación es considerar a todos los pesos de una capa como una matriz y calcular parar ella los valores singulares mínimo y máximo. Estos nos darán una idea de la variación que se producirá a la salida dada una variación a la entrada sencillamente calculando la diferencia entre estos valores. Si esta diferencia es pequeña se estaría ante una situación de insensibilidad de la capa. Una vez detectada la insensibilidad no es sencillo remediarla. Por ello esta técnica se ha usado para detener entrenamientos que a la larga serían infructuosos.

Una forma de evitar este problema es presentar un conjunto suficientemente variado de vectores de entrada. Es nuestro caso consiste en tomar una región inicial de entrenamiento suficientemente grande. De esta manera las redes se tendrán que modificar para dar la respuesta adecuada a todos los posibles valores de la entrada.

6.1.5 Ayudas al Entrenamiento

Por lo expuesto previamente es conveniente aplicar técnicas que conduzcan a las redes cerca de la trayectoria óptima. Esta acomodación se puede realizar previamente o a la vez que se aplica la función de costo de nuestro problema. De esta manera se evita la aparición de grandes comandos o tiempos negativos que conllevan fuertes correcciones. Estas ideas se concretaron en los tres procedimientos siguientes.

6.1.5.1 Entrenamiento Individual

Este procedimiento, previo al entrenamiento normal, consiste en entrenar cada red individualmente para que produzca una transición de estados concreta. Esta transición corresponderá a la de una o varias trayectorias, próximas a la óptima, para el mismo problema con el mismo número de etapas $\{\underline{x}_0, \underline{x}_1, \ldots, \underline{x}_N\}$. Habitualmente estas trayectorias

previas se obtenían con el primer método aunque también se pueden obtener a partir de los resultados de la PD.

Por este planteamiento tenemos definidos los pares de entrenamiento de cada red. Los vectores de entrada para la red i-ésima serán los puntos $\underline{x}_i$ de la trayectoria y los vectores de salida deseada los estados $\underline{x}_{i+1}$ correspondientes. Estamos, en este caso, frente a un entrenamiento supervisado. Podemos aplicar el procedimiento de entrenamiento general definiendo la función de error como el cuadrado de la distancia entre el vector obtenido y el vector deseado. En este caso no es preciso una función lineal, del tipo utilizado para eliminar los tiempos negativos, porque sólo nos interesa la proximidad a la trayectoria óptima y no la completa igualdad.

Cuando cada red está suficientemente entrenada podemos utilizarlas en conjunto para implementar la trayectoria completa. Antes de pasar al entrenamiento mediante las fórmulas se comprueba la validez del entrenamiento individual. Las trayectorias generadas por la cadena, para los mismos puntos iniciales, han de ser similares a las que se han usado para el entrenamiento.

Un inconveniente de este procedimiento es que nos basamos en trayectorias óptimas del primer método. Éstas están calculadas para una discretización de diferencia adelante. Si ahora utilizamos discretización BPF, su trayectoria óptima puede estar aún alejada de las que hemos utilizado para entrenar. Esto es debido a que los comandos que calcula la ecuación dinámica inversa del sistema pueden ser substancialmente distintos de los que se necesitan para reconstruir la trayectoria con un modelo de diferencia adelante. Los comandos o tiempos ahora necesarios pueden quedar fuera de los permitidos por lo que nos encontramos en una situación similar a la inicial.

6.1.5.2 Trayectoria Deseada

Es un procedimiento similar al anterior, porque precisa de una trayectoria, que denominamos trayectoria deseada, próxima a la óptima y con el mismo número de etapas que nuestra cadena. Para evitar el problema con la diferencia de discretizaciones, la influencia de esta trayectoria se combina, durante el entrenamiento, con la de los restantes pesos del costo como son los límites de comandos y tiempos.

El entrenamiento se realiza aplicando las fórmulas del segundo método pero sobre una función de costo modificada. En cada etapa se añade un nuevo sumando que representa la distancia entre el estado de esa etapa y el estado correspondiente de la trayectoria deseada. El peso de este sumando se va disminuyendo a medida que avanzan los entrenamientos. Cuando los resultados son razonables se elimina la influencia de la trayectoria.

6.1.5.3 Zonas Prohibidas

En los procedimientos anteriores era preciso conocer una trayectoria óptima del mismo número de etapas que nuestra cadena. Este nuevo procedimiento puede ser utilizado cuando no la podemos conseguir. Basta con realizar un estudio de las ecuaciones dinámicas del sistema o basarse en resultados de sistemas similares. Con este conocimiento determinamos una serie de zonas del espacio de estados por las que sabemos que no se van a mover las trayectorias óptimas. Una vez fijadas éstas incluimos en el costo un sumando que pese el hecho de que un estado se encuentre sobre dichas zonas y que produzca una corrección en el sentido de evitarlo.

Como vemos ésta es una restricción menos fuerte que las de los procedimientos anteriores. Sencillamente se determina una cierta banda por la que sabemos que ha de discurrir la trayectoria óptima. En cuanto todos los estados estén situados dentro de dicha banda la acción del sumando desaparecerá y tendremos únicamente la acción de la función de costo deseada. Esto concede al sistema más libertad para aproximarse finalmente a la trayectoria de menor costo.

El inconveniente de este procedimiento es que, al no estar determinado el orden relativo de los distintos estados en la trayectoria no evita los posibles bucles innecesarios y que en muchos casos son el principal inconveniente para la convergencia.

6.1.5.4 Gradiente Escalado

Otra forma de evitar el estancamiento en el aprendizaje del sistema es escalar el gradiente con factores distintos entre las distintas capas y distintas redes. Como el error se transmite de las capas finales hacia las iniciales el valor del gradiente suele disminuir a medida que nos acercamos a las capas iniciales. La proximidad a la saturación de cualquier capa intermedia hace que sean muy pequeñas las correcciones que afectan a las capas iniciales.

Para reducir el impacto de este problema, una vez determinados los delta de una capa se multiplican por un factor proporcional a la lejanía de la capa final. Por ello las capas de entrada tendrán los deltas más potenciados que las capas de salida. Este esquema se puede aplicar dentro de cada red o incluso de una red a otra, teniendo un factor mayor la capa de salida de una red que la de entrada de la siguiente.

No existe una fórmula para elegir el valor de estos factores para cada capa. Una forma sencilla, y que ha dado buenos resultados, es hacer una escala lineal. Se elige el factor máximo F que se aplicará a la primera capa y se supone un factor 1 para la capa de salida. Para una capa intermedia tendremos un factor

$$F(s)=\frac{F}{L^i}(1+L^i-s) \tag{6.29}$$

De forma similar se puede extender la fórmula si queremos aplicar la proporcionalidad al conjunto de redes. Se tomará el factor F para la capa de entrada de la primera red y se hará una escala lineal para obtener un factor *1* en la capa de salida de la última red.

6.2 Aplicación a los Problemas Ejemplo

De forma similar a como hicimos en los capítulos anteriores, a continuación vamos a presentar los resultados obtenidos para los sistemas presentados en el capítulo 3. De esta manera tendremos una visión variada de este método ante sistemas de distinta dinámica y complejidad creciente. Como en este método es posible aplicar tanto la discretización clásica como la BPF distinguiremos esos casos.

6.2.1 Primer Sistema

6.2.1.1 Expresiones Necesarias

En primer lugar debemos obtener la ecuación dinámica inversa del sistema de forma (6.1). Ésta se puede obtener utilizando discretización clásica o aplicando la formulación BPF:

a) Discretización Clásica. Utilizando diferencia adelante habíamos obtenido las ecuaciones siguientes

$$\begin{cases}(x_{1k+1}-x_{1k})/T_k = x_{2k} \\ (x_{2k+1}-x_{2k})/T_k = ax_{2k}+bx_{2k}{x_{1k}}^2+dx_{1k}+cu_k\end{cases} \tag{6.30}$$

De la primera podemos despejar la expresión para T_k

$$T_k=\frac{x_{1k+1}-x_{1k}}{x_{2k}} \tag{6.31}$$

De la segunda despejamos $\underline{u}_k$ la cual, dejándolo en función de T_k, queda

$$u_k=\left[(x_{2k+1}-x_{2k})/T_k-ax_{2k}+bx_{2k}{x_{1k}}^2+dx_{1k}\right]/c \tag{6.32}$$

Las derivadas del tipo $\partial \underline{f}^{-1}/\partial \underline{x}_k$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{x}_k}=\begin{pmatrix}\dfrac{-1}{x_{2k}} & -\dfrac{x_{1k+1}-x_{1k}}{x_{2k}^2}\end{pmatrix} \tag{6.33}$$

$$\frac{\partial u_k}{\partial x_{1k}} = \frac{1}{c}\left[-\frac{x_{2k+1} - x_{2k}}{T_k^2}\left(\frac{\partial T_k}{\partial x_{1k}}\right) - 2bx_{2k}x_{1k} - d \right] \tag{6.34}$$

$$\frac{\partial u_k}{\partial x_{2k}} = \frac{-1}{c}\left[\frac{-T_k - (x_{2k+1} - x_{2k})(\partial T_k / \partial x_{2k})}{T_k^2} - a - bx_{1k}^2 \right] \tag{6.35}$$

Las derivadas del tipo $\partial \underline{f}^{-1} / \partial \underline{x}_{k+1}$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{x}_{k+1}} = \begin{pmatrix} \dfrac{1}{x_{2k}} & 0 \end{pmatrix} \tag{6.36}$$

$$\frac{\partial u_k}{\partial \underline{x}_{k+1}} = \begin{pmatrix} -\dfrac{1}{c}\dfrac{x_{2k+1} - x_{2k}}{T_k^2}\left(\dfrac{\partial T_k}{\partial x_{1k}}\right) & \dfrac{1}{cT_k} \end{pmatrix} \tag{6.37}$$

Como en las expresiones (6.30) no aparece $\underline{v}_{k-1}$ tendremos que

$$\frac{\partial \underline{f}^{-1}}{\partial \underline{v}_{k-1}} = \mathbf{0} \tag{6.38}$$

b) Discretización BPF. Para la discretización mediante BPF debemos operar de la misma manera. Para este sistema teníamos que sus ecuaciones dinámicas quedaban

$$\begin{cases} 2(x_{1k+1} - x_{1k}) = T_k(x_{2k} + x_{2k+1}) \\ 2(x_{2k+1} - x_{2k}) = T_k\left[a(x_{2k} + x_{2k+1}) + b(x_{2k}x_{1k}^2 + x_{2k+1}x_{1k+1}^2) + d(x_{1k} + x_{1k+1}) + c(u_k + u_{k+1})\right] \end{cases} \tag{6.39}$$

Podemos asumir que los comandos $\underline{u}_k$ representan los aplicados en la etapa anterior y que los $\underline{u}_{k+1}$ son los necesarios para pasar de la etapa k a la $k+1$. Las expresiones de T_k y $\underline{u}_k$ serán

$$T_k = \frac{2(x_{1k+1} - x_{1k})}{x_{2k} + x_{2k+1}} \tag{6.40}$$

$$u_k = \frac{1}{c}\left[\frac{2(x_{2k+1} - x_{2k})}{T_k} - a(x_{2k} + x_{2k+1}) - b(x_{2k}x_{1k}^2 + x_{2k+1}x_{1k+1}^2) - d(x_{1k} + x_{1k+1}) \right] - u_{k-1} \tag{6.41}$$

Las derivadas del tipo $\partial \underline{f}^{-1} / \partial \underline{x}_k$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{x}_k} = \begin{pmatrix} \dfrac{-2}{x_{2k} + x_{2k+1}} & \dfrac{-T_k}{x_{2k} + x_{2k+1}} \end{pmatrix} \tag{6.42}$$

$$\frac{\partial u_k}{\partial x_{1k}} = \frac{1}{c}\left\{-\frac{2(x_{2k+1}-x_{2k})}{T_k^2}\left(\frac{\partial T_k}{\partial x_{1k}}\right) - b2x_{2k}x_{1k} - d\right\} \tag{6.43}$$

$$\frac{\partial u_k}{\partial x_{2k}} = \frac{1}{c}\left\{\frac{-4}{T_k}\frac{x_{2k}}{x_{2k+1}+x_{2k}} - a - bx_{1k}^2\right\} \tag{6.44}$$

Las derivadas del tipo $\partial \underline{f}^{-1}/\partial \underline{x}_{k+1}$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{x}_{k+1}} = \begin{pmatrix} \dfrac{2}{x_{2k+1}+x_{2k}} & \dfrac{-T_k}{x_{2k+1}+x_{2k}} \end{pmatrix} \tag{6.45}$$

$$\frac{\partial u_k}{\partial x_{1k+1}} = \frac{1}{c}\left\{-\frac{2(x_{2k+1}-x_{2k})}{T_k^2}\left(\frac{\partial T_k}{\partial x_{1k+1}}\right) - 2bx_{2k+1}x_{1k+1} - d\right\} \tag{6.46}$$

$$\frac{\partial u_k}{\partial x_{2k+1}} = \frac{1}{c}\left\{\frac{4}{T_k}\frac{x_{2k+1}}{x_{2k+1}+x_{2k}} - a - bx_{1k+1}^2\right\} \tag{6.47}$$

Finalmente las derivadas del tipo $\partial \underline{f}^{-1}/\partial \underline{v}_{k-1}$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{v}_{k-1}} = \begin{pmatrix} 0 & 0 \end{pmatrix} \tag{6.48}$$

$$\frac{\partial u_k}{\partial \underline{v}_{k-1}} = \begin{pmatrix} 0 & -1 \end{pmatrix} \tag{6.49}$$

Las expresiones para la función de costo y sus derivadas es independiente de la discretización elegida. Como con los métodos anteriores, queremos obtener trayectorias de tiempo mínimo con punto final el origen. Definimos entonces

$$L(\underline{x}_k, \underline{v}_k, k) = T_k \tag{6.50}$$

$$S(\underline{x}_N, N) = V_N \cdot (x_{1N}{}^2 + x_{2N}{}^2) \tag{6.51}$$

donde V_N es grande.

Las derivadas necesarias de estas funciones son

$$\frac{\partial L}{\partial \underline{x}_k} = \begin{pmatrix} \dfrac{\partial L}{\partial x_{1k}} & \dfrac{\partial L}{\partial x_{2k}} \end{pmatrix} = \begin{pmatrix} 0 & 0 \end{pmatrix} \tag{6.52}$$

$$\frac{\partial L}{\partial \underline{v}_k} = \begin{pmatrix} \dfrac{\partial L}{\partial u_k} & \dfrac{\partial L}{\partial T_k} \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix} \tag{6.53}$$

$$\frac{\partial S}{\partial \underline{x}_N} = \left(\frac{\partial S}{\partial x_{1N}} \quad \frac{\partial S}{\partial x_{2N}} \right) = \left(2V_N x_{1N} \quad 2V_N x_{2N} \right) \tag{6.54}$$

A estas expresiones es preciso añadir los sumandos para eliminar los tiempos negativos y comandos no válidos. Estos y sus derivadas tendrán la forma presentada en el apartado 6.1.3.

6.2.1.2 Resultados Obtenidos

Elegimos el valor de los parámetros del sistema igual que en los casos anteriores. Queríamos cubrir una región similar a la de PD por lo que tomamos como funciones de activación la tangente hiperbólica con factor de escala *2*. Utilizamos 7 etapas con redes de 2 capas con 5 neuronas en la capa oculta.

Para ayudar al entrenamiento aplicamos el entrenamiento individual y la trayectoria deseada. Probando de forma heurística con distintos pasos de entrenamiento para conseguir resultados aceptables.

Debemos distinguir los resultados obtenidos mediante una y otra discretización. La evolución con los entrenamientos de la trayectorias obtenidas mediante discretización clásica se muestran en la figura 6.4. Como vemos los estados se han distribuido en el tramo de bajada de la trayectoria, realizando el regreso al origen de un único paso. En cambio, a la vista de la figura 6.5, tenemos que la solución obtenida utilizando la discretización BPF distribuye los estados de uniformemente a lo largo de toda la trayectoria obteniendo mejores costos. Esto se demuestra en la figura 6.6, en la que se

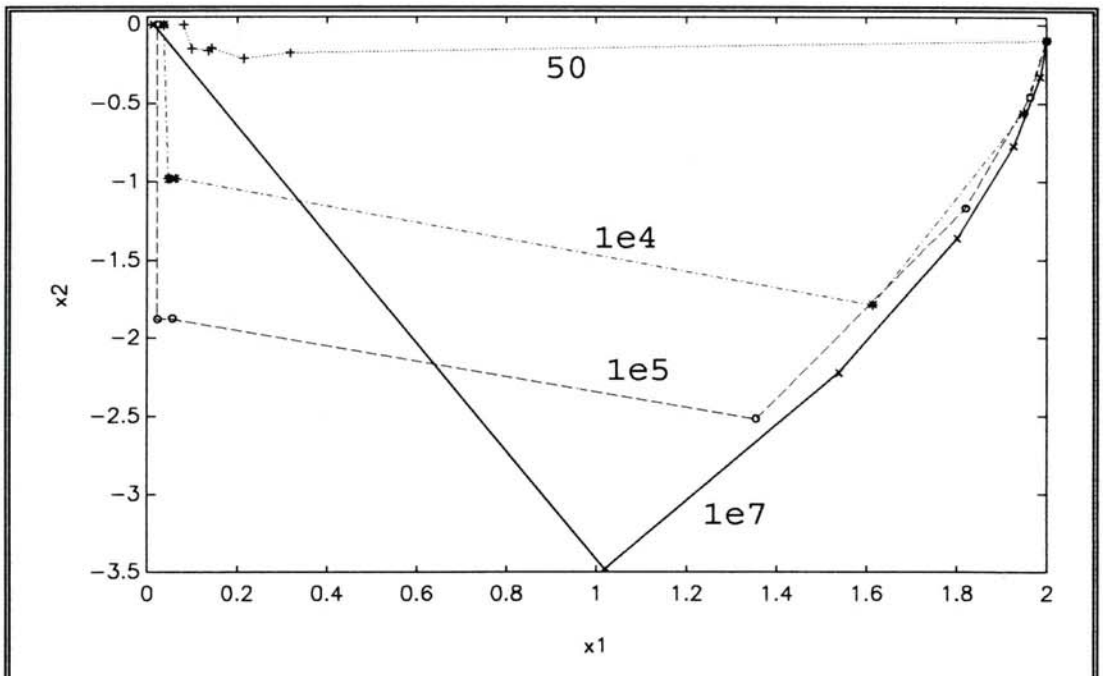

Figura 6.4 - Evolución con el entrenamiento con discretización clásica

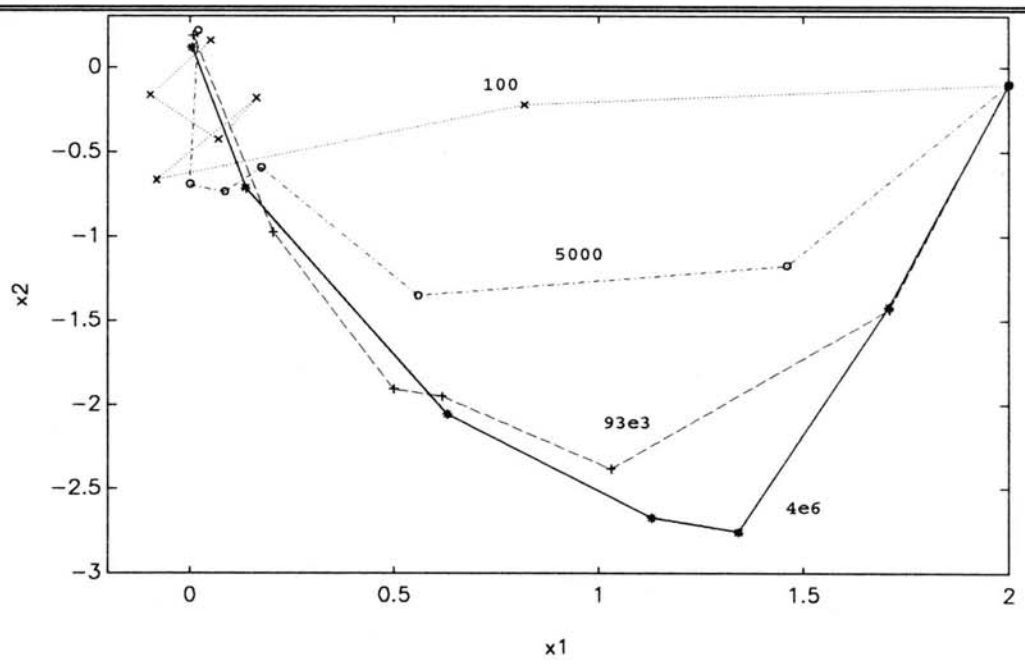

Figura 6.5 - Evolución con el entrenamiento con BPF

ve como la trayectoria de la discretización BPF, en trazo continuo, utiliza el máximo comando, llegando al origen en menos tiempo. Se demuestra así la mejora que supone la utilización de la discretización BPF en este método, al igual que ocurría en caso de la PD.

Un importante aspecto que justifica este método es el comportamiento ante

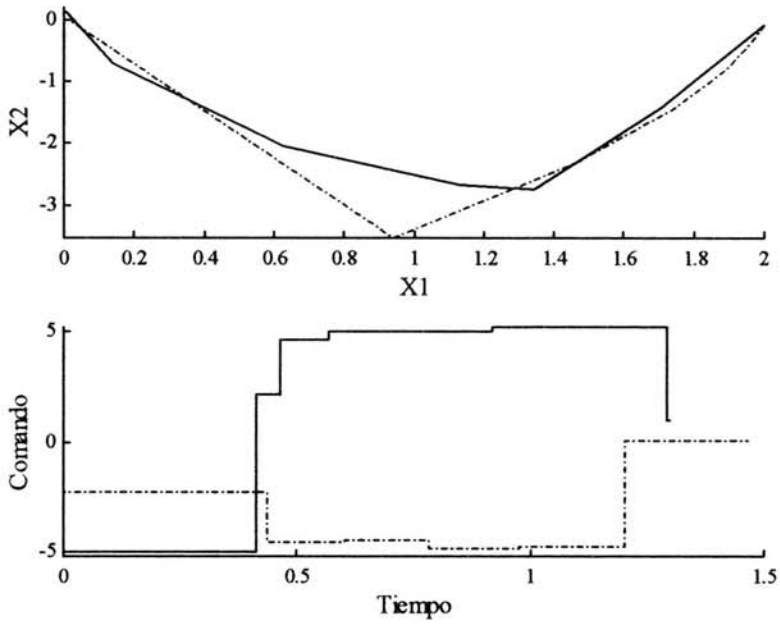

Figura 6.6 - Comparación entre resultado con discretización clásica y con BPF

perturbaciones. En la figura 6.7 vemos la comparación de una trayectoria óptima con las generadas para dos puntos iniciales que no pertenecen a la región entrenada. Como observamos estas trayectorias convergen rápidamente hacia la trayectoria óptima y llegan al punto final. Aunque ésta no representan las soluciones óptimas, ya que tardan mucho más tiempo, está asegurada la convergencia del sistema y por lo tanto la robustez del método.

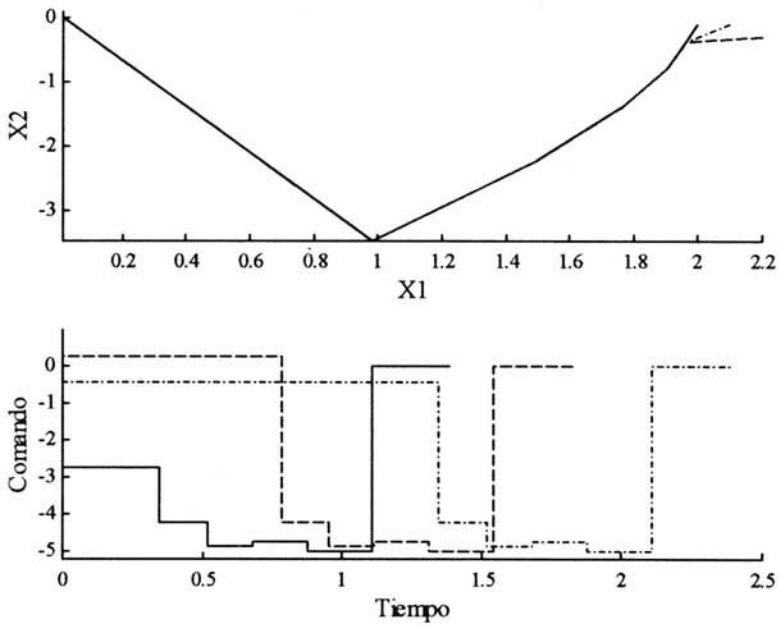

Figura 6.7 - Trayectorias para puntos fuera de la región entrenada

6.2.2 Segundo Sistema

6.2.2.1 Expresiones Necesarias

Como en el sistema anterior utilizaremos las dos discretizaciones para comparar los resultados de una y otra:

a) Discretización Clásica. Las ecuaciones dinámicas discretizadas mediante diferencia a delante son

$$\begin{cases} x_{1k+1} = x_{1k} + T_k \cos(x_{3k}) u_{1k} \\ x_{2k+1} = x_{2k} + T_k [a + \text{sen}(x_{3k}) u_{1k}] \\ x_{3k+1} = x_{3k} + T_k u_{2k} \end{cases} \tag{6.55}$$

De la segunda podemos despejar el primer comando

$$u_{1k} = \frac{x_{1k+1} - x_{1k}}{T_k \cos(x_{3k})} \tag{6.56}$$

sustituyendo este resultado en la primera y operando tenemos que

$$T_k = \frac{1}{a}\left[(x_{2k+1} - x_{2k}) - \tan(x_{3k})(x_{1k+1} - x_{1k})\right] \tag{6.57}$$

Finalmente de la tercera podemos despejar directamente

$$u_{2k} = \frac{x_{3k+1} - x_{3k}}{T_k} \tag{6.58}$$

Estas tres expresiones constituyen $\underline{f}^{I}$. Como en el sistema anterior dejamos la expresión de los comandos en función del intervalo de tiempo. De esta manera salen expresiones más sencillas y no implica ninguna limitación en el cálculo ya que basta calcular previamente esta cantidad y sustituir su valor en las expresiones.

Podemos obtener ya la expresión de las distintas derivadas. Las derivadas del tipo $\partial \underline{f}^{-1} / \partial \underline{x}_k$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{x}_k} = \left(\frac{1}{a}\tan(x_{3k}) \quad \frac{-1}{a} \quad \frac{-1}{a}\frac{(x_{1k+1} - x_{1k})}{\cos^2(x_{3k})} \right) \tag{6.59}$$

$$\frac{\partial u_{1k}}{\partial x_{1k}} = \frac{1}{\cos(x_{3k})} \frac{-T_k - (x_{1k+1} - x_{1k})(\partial T_k / \partial x_{1k})}{T_k^{\,2}} \tag{6.60}$$

$$\frac{\partial u_{1k}}{\partial x_{2k}} = -\frac{x_{1k+1} - x_{1k}}{\cos(x_{3k})} \frac{(\partial T_k / \partial x_{2k})}{T_k^{\,2}} \tag{6.61}$$

$$\frac{\partial u_{1k}}{\partial x_{3k}} = -(x_{1k+1} - x_{1k}) \frac{(\partial T_k / \partial x_{3k})\cos(x_{3k}) - T_k \,\mathrm{sen}(x_{3k})}{T_k^{\,2}\cos^2(x_{3k})} \tag{6.62}$$

$$\frac{\partial u_{2k}}{\partial x_{1k}} = -(x_{3k+1} - x_{3k}) \frac{(\partial T_k / \partial x_{1k})}{T_k^{\,2}} \tag{6.63}$$

$$\frac{\partial u_{2k}}{\partial x_{2k}} = -(x_{3k+1} - x_{3k}) \frac{(\partial T_k / \partial x_{2k})}{T_k^{\,2}} \tag{6.64}$$

$$\frac{\partial u_{2k}}{\partial x_{3k}} = \frac{-1 - u_{2k}(\partial T_k / \partial x_{3k})}{T_k} \tag{6.65}$$

Las derivadas del tipo $\partial \underline{f}^{-1} / \partial \underline{x}_{k+1}$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{x}_{k+1}} = \begin{pmatrix} \frac{-1}{a}\tan(x_{3k}) & \frac{1}{a} & 0 \end{pmatrix} \tag{6.66}$$

$$\frac{\partial u_{1k}}{\partial x_{1k+1}} = \frac{1}{\cos(x_{3k})} \frac{T_k - (x_{1k+1} - x_{1k})(\partial T_k / \partial x_{1k+1})}{T_k^{\,2}} \tag{6.67}$$

$$\frac{\partial u_{1k}}{\partial x_{2k+1}} = -\frac{x_{1k+1} - x_{1k}}{\cos(x_{3k})} \frac{(\partial T_k / \partial x_{2k+1})}{T_k^{\,2}} \tag{6.68}$$

$$\frac{\partial u_{1k}}{\partial x_{3k+1}} = 0 \tag{6.69}$$

$$\frac{\partial u_{2k}}{\partial x_{1k+1}} = -(x_{3k+1} - x_{3k}) \frac{(\partial T_k / \partial x_{1k+1})}{T_k^{\,2}} \tag{6.70}$$

$$\frac{\partial u_{2k}}{\partial x_{2k+1}} = -(x_{3k+1} - x_{3k}) \frac{(\partial T_k / \partial x_{2k+1})}{T_k^{\,2}} \tag{6.71}$$

$$\frac{\partial u_{2k}}{\partial x_{3k+1}} = \frac{1}{T_k} \tag{6.72}$$

Ya que en las expresiones no aparece $\underline{v}_{k-1}$ tendremos que

$$\partial \underline{v}_k / \partial \underline{v}_{k-1} = \mathbf{0} \tag{6.73}$$

b) Discretización BPF. Para la formulación BPF tenemos

$$\begin{cases} 2(x_{1k+1} - x_{1k})/T_k = \cos(x_{3k})u_{1k-1} + \cos(x_{3k+1})u_{1k} \\ 2(x_{2k+1} - x_{2k})/T_k = a + \text{sen}(x_{3k})u_{1k-1} + \text{sen}(x_{3k+1})u_{1k} \\ 2(x_{3k+1} - x_{3k})/T_k = u_{2k-1} + u_{2k} \end{cases} \tag{6.74}$$

Donde, en este caso, los comandos implicados son los de la etapa anterior y la actual. Operando con estas ecuaciones podemos obtener las tres componentes de $\underline{f}^1$:

$$u_{1k} = \frac{1}{\cos(x_{3k+1})} \left[2\frac{x_{1k+1} - x_{1k}}{T_k} - \cos(x_{3k})u_{1k-1} \right] \tag{6.75}$$

$$u_{2k} = 2\frac{x_{3k+1} - x_{3k}}{T_k} - u_{2k-1} \tag{6.76}$$

$$T_k = 2\frac{(x_{2k+1} - x_{2k}) - \tan(x_{3k+1})(x_{1k+1} - x_{1k})}{2a + \text{sen}(x_{3k})u_{1k-1} - \tan(x_{3k+1})\cos(x_{3k})u_{1k-1}} \tag{6.77}$$

Las derivadas del tipo $\partial \underline{f}^{-1}/\partial \underline{x}_k$ son las siguientes:

$$\frac{\partial T_k}{\partial x_{1k}} = \frac{2\tan(x_{3k+1})}{2a + \text{sen}(x_{3k})u_{1k} - \tan(x_{3k+1})\cos(x_{3k})u_{1k-1}} \tag{6.78}$$

$$\frac{\partial T_k}{\partial x_{2k}} = \frac{-2}{2a + \text{sen}(x_{3k})u_{1k-1} - \tan(x_{3k+1})\cos(x_{3k})u_{1k-1}} \tag{6.79}$$

$$\frac{\partial T_k}{\partial x_{3k}} = -T_k u_{1k-1} \frac{\cos(x_{3k}) + \tan(x_{3k+1})\text{sen}(x_{3k})}{2a + \text{sen}(x_{3k})u_{1k-1} - \tan(x_{3k+1})\cos(x_{3k})u_{1k-1}} \tag{6.80}$$

$$\frac{\partial u_{1k}}{\partial x_{1k}} = \frac{2}{\cos(x_{3k+1})} \cdot \frac{-T_k - (x_{1k+1} - x_{1k})(\partial T_k/\partial x_{1k})}{T_k^{\,2}} \tag{6.81}$$

$$\frac{\partial u_{1k}}{\partial x_{2k}} = \frac{2(x_{1k+1} - x_{1k})}{\cos(x_{3k+1})} \cdot \frac{-(\partial T_k/\partial x_{2k})}{T_k^{\,2}} \tag{6.82}$$

$$\frac{\partial u_{1k}}{\partial x_{3k}} = \frac{1}{\cos(x_{3k+1})}\left[-2\frac{x_{1k+1} - x_{1k}}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial x_{3k}}\right) + \text{sen}(x_{3k})u_{1k-1}\right] \tag{6.83}$$

$$\frac{\partial u_{2k}}{\partial x_{1k}} = -2\frac{x_{3k+1} - x_{3k}}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial x_{1k}}\right) \tag{6.84}$$

$$\frac{\partial u_{2k}}{\partial x_{2k}} = -2\frac{x_{3k+1} - x_{3k}}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial x_{2k}}\right) \tag{6.85}$$

$$\frac{\partial u_{2k}}{\partial x_{3k}} = 2\frac{-T_k - (x_{3k+1} - x_{3k})(\partial T_k/\partial x_{3k})}{T_k^{\,2}} \tag{6.86}$$

Las derivadas del tipo $\partial \underline{f}^{-1}/\partial \underline{x}_{k+1}$ son las siguientes:

$$\frac{\partial T_k}{\partial x_{1k+1}} = \frac{-2\tan(x_{3k+1})}{2a + \text{sen}(x_{3k})u_{1k-1} - \tan(x_{3k+1})\cos(x_{3k})u_{1k-1}} \tag{6.87}$$

$$\frac{\partial T_k}{\partial x_{2k+1}} = \frac{2}{2a + \text{sen}(x_{3k})u_{1k-1} - \tan(x_{3k+1})\cos(x_{3k})u_{1k-1}} \tag{6.88}$$

$$\frac{\partial T_k}{\partial x_{3k+1}} = \frac{1}{\cos^2(x_{3k+1})} \cdot \frac{-2(x_{1k+1} - x_{1k}) + T_k\cos(x_{3k})u_{1k-1}}{2a + \text{sen}(x_{3k})u_{1k-1} - \tan(x_{3k+1})\cos(x_{3k})u_{1k-1}} \tag{6.89}$$

$$\frac{\partial u_{1k}}{\partial x_{1k+1}} = \frac{2}{\cos(x_{3k+1})} \cdot \frac{T_k - (x_{1k+1} - x_{1k})(\partial T_k/\partial x_{1k+1})}{T_k^{\,2}} \tag{6.90}$$

$$\frac{\partial u_{1k}}{\partial x_{2k+1}} = \frac{2(x_{1k+1} - x_{1k})}{\cos(x_{3k+1})} \cdot \frac{-\left(\partial T_k / \partial x_{2k+1}\right)}{T_k^{\,2}} \tag{6.91}$$

$$\frac{\partial u_{1k}}{\partial x_{3k+1}} = \frac{1}{\cos(x_{3k+1})}\left[-2\frac{(x_{1k+1} - x_{1k})\left(\partial T_k / \partial x_{3k+1}\right)}{T_k^{\,2}} + u_{1k}\,\mathrm{sen}(x_{3k+1})\right] \tag{6.92}$$

$$\frac{\partial u_{2k}}{\partial x_{1k+1}} = -2\frac{x_{3k+1} - x_{3k}}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial x_{1k+1}}\right) \tag{6.93}$$

$$\frac{\partial u_{2k}}{\partial x_{2k+1}} = -2\frac{x_{3k+1} - x_{3k}}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial x_{2k+1}}\right) \tag{6.94}$$

$$\frac{\partial u_{2k}}{\partial x_{3k+1}} = 2\frac{T_k - (x_{3k+1} - x_{3k})\left(\partial T_k / \partial x_{3k+1}\right)}{T_k^{\,2}} \tag{6.95}$$

Finalmente las derivadas del tipo $\partial \underline{f}^{-1} \big/ \partial \underline{v}_{k-1}$ son las siguientes:

$$\frac{\partial T_k}{\partial u_{1k-1}} = -2T_k\,\frac{\mathrm{sen}(x_{3k}) - \tan(x_{3k+1})\cos(x_{3k})}{2a + \mathrm{sen}(x_{3k})u_{1k-1} - \tan(x_{3k+1})\cos(x_{3k})u_{1k-1}} \tag{6.96}$$

$$\frac{\partial T_k}{\partial u_{2k-1}} = 0 \tag{6.97}$$

$$\frac{\partial u_{1k}}{\partial u_{1k-1}} = \frac{1}{\cos(x_{3k+1})}\left[-2\frac{x_{1k+1} - x_{1k}}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial u_{1k-1}}\right) - \cos(x_{3k})\right] \tag{6.98}$$

$$\frac{\partial u_{1k}}{\partial u_{2k-1}} = 0 \tag{6.99}$$

$$\frac{\partial u_{2k}}{\partial u_{1k-1}} = -2\frac{x_{3k+1} - x_{3k}}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial u_{1k-1}}\right) \tag{6.100}$$

$$\frac{\partial u_{2k}}{\partial u_{2k-1}} = -1 \tag{6.101}$$

y, ya que T_{k-1} no aparece en $\underline{f}^{-1}$, finalmente

$$\frac{\partial \underline{v}_k}{\partial T_{k-1}} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^{\mathrm{T}} \tag{6.102}$$

En cuanto a las funciones de costo, como deseamos trayectorias de tiempo mínimo y que el sistema llegue al origen sin importar la orientación, quedarán definidas

$$L(\underline{x}_k, \underline{v}_k, k) = T_k \tag{6.103}$$

$$S(\underline{x}_N, N) = V_N \cdot (x_{1N}{}^2 + x_{2N}{}^2) \tag{6.104}$$

Sus derivadas son, por lo tanto

$$\frac{\partial L}{\partial \underline{x}_k} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \tag{6.105}$$

$$\frac{\partial L}{\partial \underline{v}_k} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{6.106}$$

$$\frac{\partial S}{\partial \underline{x}_N} = \begin{pmatrix} 2V_N x_{1N} & 2V_N x_{2N} & 0 \end{pmatrix} \tag{6.107}$$

Debemos añadir las funciones *L* necesarias para eliminar tiempos negativos y comandos no admitidos, así como aquellas precisas para acelerar la convergencia.

6.2.2.2 Resultados

Como con los métodos anteriores, tomamos un valor para el parámetro *a=0.01*. De la misma manera que en el método primero elegimos todas las redes iguales con una capa oculta de 5 neuronas. Las funciones de activación para la capa de salida son: la primera sigmoide escalada por *10*, ya que estamos suponiendo que existe una pared que no permite pasar a valores de $x_1<0$; la segunda y tercera tangentes hiperbólicas escaladas por *10*. En este caso la cota para el avance, fijada en la función de costo, es de la unidad, mientras que para el giro es de *10*.

Es necesario ayudar al proceso de entrenamiento. Las técnicas más eficientes son las de escalado del gradiente a todas las redes, con un factor en torno a *2* para la primera

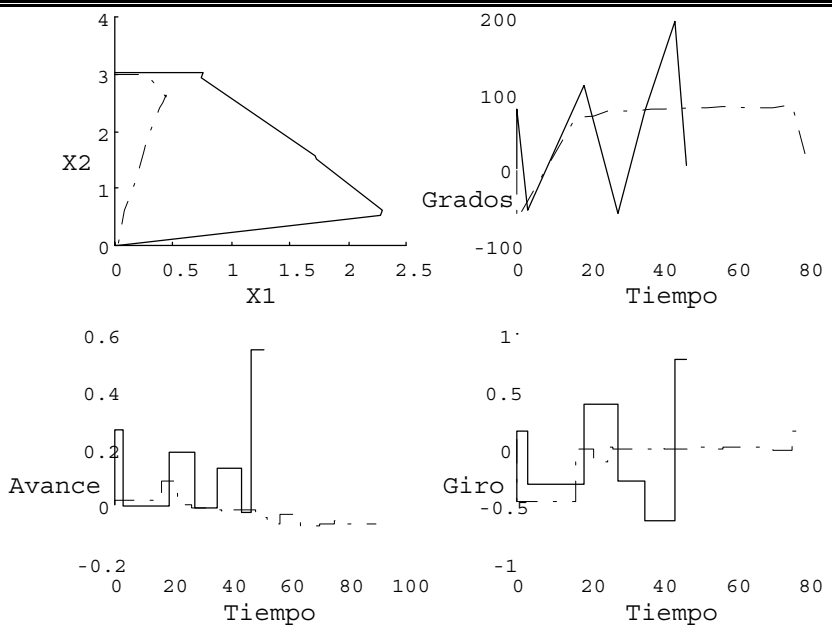

Figura 6.8 - Comparación de discretización clásica y BPF.

capa de la primera red. También da buenos resultados el uso, como trayectoria deseada, de una obtenida mediante el método primero. Además, el algoritmo utilizado es el Fletcher-Reeves y una acumulación de 10 gradientes antes de pasar a corregir los pesos.

Debemos distinguir los resultados obtenidos mediante una y otra discretización porque son significativamente diferentes. Vemos, en la figura 6.8, que la discretización clásica (trazo discontinuo) se separa mucho menos del eje y aplica menos comando, por lo que la trayectoria tarda más. En el caso de la discretización BPF (trazo continuo) la trayectoria se aleja del eje y utiliza más comando con lo que consigue llegar al origen en menos tiempo. Esto ocurre a pesar de que para discretización clásica estamos utilizando 15 etapas mientras que para la BPF sólo 7. Esto es debido a la dificultad de entrenamiento en este segundo caso que hacen necesario reducir el número de redes para conseguir la convergencia.

Un importante aspecto es el comportamiento ante perturbaciones. En la figura 6.9 vemos la trayectoria para distintos puntos iniciales que no pertenecen a la región entrenada. Como observamos estas trayectorias convergen rápidamente hacia la trayectoria óptima. Se observa que son trayectorias subóptimas, ya que tardan mucho más tiempo, pero llegan al punto final. Esto asegura la convergencia del sistema y por lo tanto la robustez de este método.

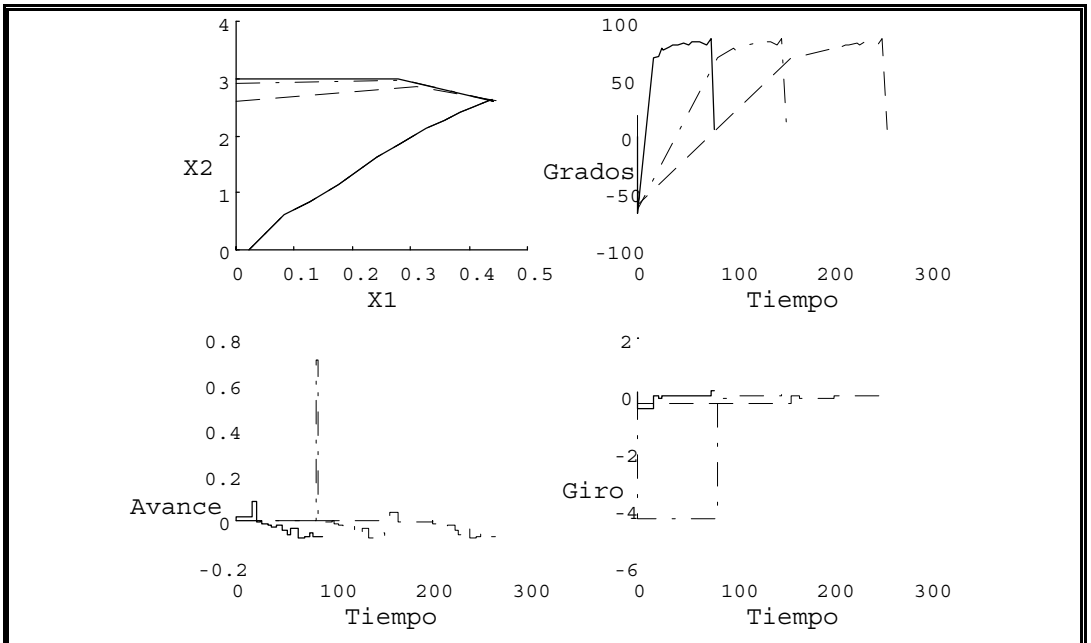

Figura 6.9 - Perturbación del punto inicial.

6.2.3 Tercer Sistema

6.2.3.1 Expresiones Necesarias

a) Discretización Clásica. Las fórmulas discretizadas mediante diferencia adelante para este sistema, obtenidas en el capítulo 3, son las siguientes

$$\begin{cases} h_{1k+1} = h_{1k} + T_k\left[-R_1\Delta h_k + q_{lk}\right] \\ h_{2k+1} = h_{2k} + T_k\left[R_1\Delta h_k - R_2 h_{2k}\right] \\ \theta_{k+1} = \theta_k + T_k\left\{-\theta_k\left[1/R_T + C_p\left(R_1\Delta h_k - R_2 h_{2k}\right)\right] + q_{ck}\right\}/C_s \end{cases} \tag{6.108}$$

donde hemos tomado *P(x)=x*. Despejando los comandos en función del estado actual y el siguiente tendremos que

$$T_k = \frac{h_{2k+1} - h_{2k}}{R_1\Delta h_k - R_2 h_{2k}} \tag{6.109}$$

$$q_{lk} = \frac{h_{1k+1} - h_{1k}}{T_k} + R_1\Delta h_k \tag{6.110}$$

$$q_{ck} = C_s\frac{\theta_{k+1} - \theta_k}{T_k} + \theta_k\left[\frac{1}{R_T} + C_p\left(R_1\Delta h_k - R_2 h_{2k}\right)\right] \tag{6.111}$$

Las derivadas del tipo $\partial \underline{f}^{-1}/\partial \underline{x}_k$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{x}_k} = \begin{pmatrix} -\dfrac{T_k R_1}{R_1\Delta h_k - R_2 h_{2k}} & \dfrac{T_k(R_1+R_2)-1}{R_1\Delta h_k - R_2 h_{2k}} & 0 \end{pmatrix} \tag{6.112}$$

$$\frac{\partial q_{lk}}{\partial \underline{x}_k} = \begin{pmatrix} \dfrac{-T_k - \left(h_{1k+1} - h_{1k}\right)\left(\partial T_k/\partial h_{1k}\right)}{T_k^{\,2}} + R_1 & \dfrac{-\left(h_{1k+1} - h_{1k}\right)}{T_k^{\,2}}\left(\dfrac{\partial T_k}{\partial h_{2k}}\right) - R_1 & 0 \end{pmatrix} \tag{6.113}$$

$$\frac{\partial q_{ck}}{\partial h_{1k}} = -C_s\frac{\theta_{k+1} - \theta_k}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial h_{1k}}\right) + \theta_k C_p R_1 \tag{6.114}$$

$$\frac{\partial q_{ck}}{\partial h_{2k}} = -C_s\frac{\theta_{k+1} - \theta_k}{T_k^{\,2}}\left(\frac{\partial T_k}{\partial h_{2k}}\right) + \theta_k C_p(-R_1 - R_2) \tag{6.115}$$

$$\frac{\partial q_{ck}}{\partial \theta_k} = -\frac{C_s}{T_k} + \left[\frac{1}{R_T} + C_p\left(R_1\Delta h_k - R_2 h_{2k}\right)\right] \tag{6.116}$$

Las derivadas del tipo $\partial \underline{f}^{-1}/\partial \underline{x}_{k+1}$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{x}_{k+1}} = \begin{pmatrix} 0 & \dfrac{1}{R_1\Delta h_k - R_2 h_{2k}} & 0 \end{pmatrix} \tag{6.117}$$

$$\frac{\partial q_{lk}}{\partial \underline{x}_{k+1}} = \begin{pmatrix} \dfrac{1}{T_k} & \dfrac{-(h_{1k+1} - h_{1k})}{T_k^{\,2}} \left(\dfrac{\partial T_k}{\partial h_{2k+1}}\right) & 0 \end{pmatrix} \tag{6.118}$$

$$\frac{\partial q_{ck}}{\partial \underline{x}_{k+1}} = \begin{pmatrix} 0 & -C_s \dfrac{\theta_{k+1} - \theta_k}{T_k^{\,2}} \left(\dfrac{\partial T_k}{\partial h_{2k+1}}\right) & \dfrac{C_s}{T_k} \end{pmatrix} \tag{6.119}$$

Como en las expresiones no aparece $\underline{v}_{k-1}$ tendremos que

$$\frac{\partial \underline{f}^{-1}}{\partial \underline{v}_{k-1}} = \mathbf{0} \tag{6.120}$$

b) Discretización BPF. La discretización BPF para este sistema, considerando los comandos de la etapa anterior y actual, se puede escribir

$$2(h_{1k+1} - h_{1k})/T_k = \left[-R_1 P(\Delta h_k) + q_{lk-1}\right] + \left[-R_1 P(\Delta h_{k+1}) + q_{lk}\right] \tag{6.121}$$

$$2(h_{2k+1} - h_{2k})/T_k = \left[R_1 P(\Delta h_k) - R_2 P(h_{2k})\right] + \left[R_1 P(\Delta h_{k+1}) - R_2 P(h_{2k+1})\right] \tag{6.122}$$

$$\begin{aligned} 2(\theta_{k+1} - \theta_k)/T_k = {} & \left\{-\theta_k\left[1/R_T + C_p\left(R_1 P(\Delta h_k) - R_2 P(h_{2k})\right)\right] + q_{ck-1}\right\}\Big/ C_s + \\ & + \left\{-\theta_{k+1}\left[1/R_T + C_p\left(R_1 P(\Delta h_{k+1}) - R_2 P(h_{2k+1})\right)\right] + q_{ck}\right\}\Big/ C_s \end{aligned} \tag{6.123}$$

Despejando el intervalo de tiempo y los comandos quedarán:

$$T_k = 2\frac{h_{2k+1} - h_{2k}}{R_1(\Delta h_k + \Delta h_{k+1}) - R_2(h_{2k} + h_{2k+1})} \tag{6.124}$$

$$q_{lk} = 2\frac{h_{1k+1} - h_{1k}}{T_k} + R_1\left(\Delta h_k + \Delta h_{k+1}\right) - q_{lk-1} \tag{6.125}$$

$$\begin{aligned} q_{ck} = 2\frac{\theta_{k+1} - \theta_k}{T_k} + \frac{1}{C_s}\Bigg\{ & \frac{\theta_k + \theta_{k+1}}{R_T} + C_p R_1\left[\theta_k \Delta h_k + \theta_{k+1}\Delta h_{k+1}\right] \\ & - C_p R_2\left[\theta_k h_{2k} + \theta_{k+1} h_{2k+1}\right]\Bigg\} - q_{ck-1} \end{aligned} \tag{6.126}$$

Las derivadas del tipo $\partial \underline{f}^{-1} \big/ \partial \underline{x}_k$ son las siguientes:

$$\frac{\partial T_k}{\partial h_{1k}} = \frac{-R_1 T_k}{R_1(\Delta h_k + \Delta h_{k+1}) - R_2(h_{2k} + h_{2k+1})} \tag{6.127}$$

$$\frac{\partial T_k}{\partial h_{2k}} = 2\frac{-R_1(\Delta h_k + \Delta h_{k+1}) - R_1 h_{2k} + (R_1 + 2R_2)h_{2k+1}}{\left[R_1(\Delta h_k + \Delta h_{k+1}) - R_2(h_{2k} + h_{2k+1})\right]^2} \tag{6.128}$$

$$\frac{\partial q_{lk}}{\partial h_{1k}} = 2\frac{-T_k - (h_{1k+1} - h_{1k})\left(\partial T_k / \partial h_{1k}\right)}{T_k^2} + R_1 \tag{6.129}$$

$$\frac{\partial q_{lk}}{\partial h_{2k}} = -2\frac{h_{1k+1} - h_{1k}}{T_k^2}\left(\frac{\partial T_k}{\partial h_{2k}}\right) - R_1 \tag{6.130}$$

$$\frac{\partial q_{ck}}{\partial h_{1k}} = -2\frac{\theta_{k+1} - \theta_k}{T_k^2}\left(\frac{\partial T_k}{\partial h_{1k}}\right) + \frac{C_p R_1}{C_s}\theta_k \tag{6.131}$$

$$\frac{\partial q_{ck}}{\partial h_{2k}} = -2\frac{\theta_{k+1} - \theta_k}{T_k^2}\left(\frac{\partial T_k}{\partial h_{2k}}\right) - \frac{C_p \theta_k}{C_s}(R_1 + R_2) \tag{6.132}$$

$$\frac{\partial \underline{v}_k}{\partial \theta_k} = \left(0 \quad 0 \quad \frac{-2}{T_k} + \frac{1}{C_s}\left[\frac{1}{R_T} + C_p(R_1 \Delta h_k - R_2 h_{2k})\right]\right)^{\mathrm{T}} \tag{6.133}$$

Las derivadas del tipo $\partial \underline{f}^{-1} / \partial \underline{x}_{k+1}$ son las siguientes:

$$\frac{\partial T_k}{\partial h_{1k+1}} = \frac{-R_1 T_k}{R_1(\Delta h_k + \Delta h_{k+1}) - R_2(h_{2k} + h_{2k+1})} \tag{6.134}$$

$$\frac{\partial T_k}{\partial h_{2k+1}} = 2\frac{R_1(\Delta h_k + \Delta h_{k+1}) - (R_1 + 2R_2)h_{2k} + R_1 h_{2k+1}}{\left[R_1(\Delta h_k + \Delta h_{k+1}) - R_2(h_{2k} + h_{2k+1})\right]^2} \tag{6.135}$$

$$\frac{\partial q_{lk}}{\partial h_{1k+1}} = 2\frac{T_k - (h_{1k+1} - h_{1k})\left(\partial T_k / \partial h_{1k+1}\right)}{T_k^2} + R_1 \tag{6.136}$$

$$\frac{\partial q_{lk}}{\partial h_{2k+1}} = -2\frac{h_{1k+1} - h_{1k}}{T_k^2}\left(\frac{\partial T_k}{\partial h_{2k+1}}\right) - R_1 \tag{6.137}$$

$$\frac{\partial q_{ck}}{\partial h_{1k+1}} = -2\frac{\theta_{k+1} - \theta_k}{T_k^2}\left(\frac{\partial T_k}{\partial h_{1k+1}}\right) + \frac{C_p R_1}{C_s}\theta_k \tag{6.138}$$

$$\frac{\partial q_{ck}}{\partial h_{2k+1}} = -2\frac{\theta_{k+1} - \theta_k}{T_k^2}\left(\frac{\partial T_k}{\partial h_{2k+1}}\right) - \frac{C_p \theta_k}{C_s}(R_1 + R_2) \tag{6.139}$$

$$\frac{\partial \underline{v}_k}{\partial \theta_{k+1}} = \left(0 \quad 0 \quad \frac{2}{T_k} + \frac{1}{C_s}\left[\frac{1}{R_T} + C_p(R_1 \Delta h_k - R_2 h_{2k})\right]\right)^{\mathrm{T}} \tag{6.140}$$

Finalmente las derivadas del tipo $\partial \underline{f}^{-1} / \partial \underline{v}_{k-1}$ son las siguientes:

$$\frac{\partial T_k}{\partial \underline{v}_{k-1}} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \tag{6.141}$$

$$\frac{\partial q_{lk}}{\partial \underline{v}_{k-1}} = \begin{pmatrix} -1 & 0 & 0 \end{pmatrix} \tag{6.142}$$

$$\frac{\partial q_{ck}}{\partial \underline{v}_{k-1}} = \begin{pmatrix} 0 & -1 & 0 \end{pmatrix} \tag{6.143}$$

Las expresiones de la función de costo son comunes a ambas discretizaciones. Para trayectorias de tiempo mínimo

$$L(\underline{x}_k, \underline{v}_k, k) = T_k \tag{6.144}$$

por lo que

$$\frac{\partial L}{\partial \underline{x}_k} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix} \tag{6.145}$$

$$\frac{\partial L}{\partial \underline{v}_k} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{6.146}$$

Para alcanzar el punto final (h_1^f, h_2^f, θ^f), consecuencia de fijar el caudal de salida y la temperatura del segundo depósito, definiremos la función *S* de la siguiente manera

$$S(\underline{x}_N, N) = V_N \cdot \left[(h_1^f - h_{1N})^2 + (h_2^f - h_{2N})^2 + (\theta^f - \theta_N)^2 \right] \tag{6.147}$$

Sus derivadas son

$$\frac{\partial S}{\partial \underline{x}_N} = \begin{pmatrix} 2V_N h_{1N} & 2V_N h_{2N} & 2V_N \theta_N \end{pmatrix} \tag{6.148}$$

6.2.3.2 Resultados

Fijamos los parámetros a los mismos valores que utilizamos para la PD. Las alturas sólo pueden variar en el rango de *0* a *10*, por ello elegimos como función de activación para las dos primeras neuronas de la capa de salida una sigmoide escalada por *10*. En cuanto a la variación de temperatura, ésta sólo podrá ser positiva ya que calentamos el líquido del segundo depósito. Elegimos entonces también una sigmoide con escala *10* para la tercera neurona de la capa de salida. Para el punto final fijamos unos valores de $h_2^f=5$ y $\theta^f=5$; por este valor de h_1^f y por los de R_1 y R_2 tenemos que ha de ser $h_1^f=5$.

Con estas condiciones, tanto con la discretización clásica como con la BPF, hemos ensayado numerosas configuraciones en cuanto a número de etapas, capas de las redes y número de neuronas en las capas. En cada una de ellas hemos probado todas las

variantes del algoritmo de entrenamiento y hemos utilizado todas las técnicas de aceleración descritas en este capítulo. En cada uno de los casos hemos hecho numerosos ensayos variando, de forma supervisada, los parámetros disponibles en el algoritmo de entrenamiento. Incluso en los mejores casos, la evolución del entrenamiento ha sido desesperadamente lento teniéndose que abandonar la mayoría de ellos. Por este motivo, hasta el momento de finalización de escritura de esta memoria, no se han obtenido resultados dignos de mención.

6.2.4 Resumen y Conclusiones

De la aplicación de este segundo método a los problemas ejemplo podemos concluir las siguientes características:

- Los resultados obtenidos difieren en base a la discretización elegida. En el caso de utilizar BPF, ésta consigue resultamos mejores que la discretización clásica. Se reafirma con ello las buenas propiedades que esta formulación había demostrado en la PD.
- En caso de perturbaciones, este método consigue que el sistema vuelva a la trayectoria óptima. Aunque, desde ese punto perturbado esta solución es subóptima, se consigue que el sistema llegue al punto final deseado. Por esta razón se trata de un método robusto. Cumple entonces el objetivo de superar el problema del primer método de RN.
- El entrenamiento de la cadena de RN es muy largo y difícil. Se emplearon numerosas técnicas y modificaciones del algoritmo para conseguir la convergencia. A pesar de ello, para el problema más complejo no fue posible obtener resultados satisfactorios. En cualquier caso, los problemas de entrenamiento que sufre, aunque son importantes, afectan únicamente en el proceso previo, y no en su utilización.

Se puede concluir que este método es mejor que el primero ya que es robusto y permite la utilización de la BPF. En sistemas de dinámica compleja, debemos todavía mejorar las técnicas de aprendizaje para que el proceso converja.

7

Implementación

En este capítulo presentaremos la forma en que han sido realizados los distintos estudios. Los algoritmos han sido codificados en C y en MATLAB para poder portarse a cualquier tipo de ordenador. En primer lugar veremos las características de los sistemas informáticos que se usaron en la implementación. En segundo lugar veremos las estructuras de datos y la forma en que se codificaron los algoritmos de los distintos métodos. También haremos mención a la forma en que se trataron y visualizaron los resultados.

7.1 Características Generales

En este apartado veremos las decisiones tomadas en cuanto a la implementación de los algoritmos en sus diferentes aspectos.

7.1.1 Sistema Operativo

La principal característica de los problemas tratados en cuanto a su resolución es la importante complejidad espacial y temporal. La complejidad espacial hace necesaria la

utilización de estructuras de datos de gran tamaño. Como destacamos en su momento, en algunos problemas de la PD se presentaban requerimientos de memoria que estaban en torno a los 30 Mb. No son usuales ordenadores con esa cantidad de memoria, o al menos que puedan ponerla a disposición de un único programa. Se hace imprescindible, entonces, que el sistema operativo posea memoria virtual. El gran número de cálculos hace que los programas duren largos periodos de tiempo. Es por ello es conveniente un sistema operativo (SO) que permita la utilización de la máquina con otros procesos de menor carga y que a su vez sea resistente a fallos. Se ha de tratar pues de un SO multiproceso robusto.

A la vista de estas características optamos por realizar los cálculo bajo el sistema operativo UNIX. Concretamente, al disponer de ordenadores de la marca Hewlett-Packar: HP-9000 serie 340 y HP-9000 serie Apolo-700, utilizamos la versión HP-UX [Hewl 92]. Este SO posee otras características importantes. La estructura de su sistema de ficheros permite un cómodo almacenamiento de los datos. La posibilidad de preparación de trabajos, a corto y medio plazo, mediante las colas de ejecución, consigue un rendimiento ininterrumpido del sistema. También, por ser uno de los SO más extendidos, existen infinidad de utilidades que abren un amplio abanico de posibilidades.

7.1.2 Lenguaje de Programación

Se eligió el lenguaje de programación *C* para la codificación de nuestros algoritmos [Kern 88]. Éste es un lenguaje de propósito general que da gran libertad al programador, permitiéndole trabajar tanto en alto nivel como entrar en los detalles de bajo nivel. No posee grandes librerías para cálculo pero, por la sencillez de las operaciones a realizar en nuestro caso, no son necesarias. En cambio, una gran virtud que posee es la eficiencia de manejo de estructuras en memoria. La utilización de los punteros permite, si el programador tiene cierta disciplina, manejar, de la manera más eficiente y estructurada posible, complejas estructuras de datos. En nuestro caso esto ha sido una baza fundamental, sobre todo al tratar con las cadenas de redes neuronales, ya que la estructura, en principio, puede ser distinta en cada una de ellas.

Otra ventaja es que, al tratarse de un lenguaje muy extendido en el entorno UNIX, los compiladores poseen un alto grado de optimización que aprovecha al máximo las posibilidades del repertorio de cada procesador. Además existen numerosas herramientas de depuración y mantenimiento que facilitan la corrección de errores.

Su principal inconveniente es su escasa legibilidad. Para que pueda ser entendida con facilidad, ciertas partes del código han de ser comentadas con profusión. Esto dificulta también, en ciertas ocasiones, la modificación y ampliación del código. A la vista de

estos inconvenientes sería recomendable en un futuro la utilización de la extensión a este lenguaje, denominada *C++*, que permite la programación orientada a objetos. Este lenguaje facilita la ampliación y sobre todo la reutilización de partes críticas del código que ya han sido costosamente depuradas.

7.1.3 Estudio de los Resultados

A priori perece que, por tratarse de problemas de control óptimo, únicamente en el resultado se debe considerar el costo. Evidentemente esta magnitud no es suficiente para la comparación de resultados y sobre todo para detectar los fallos e introducir mejoras. Se hace necesario trabajar, al menos, con la secuencia de estados y comandos que constituyen la trayectoria óptima. La mejor forma de representar estos datos es gráficamente. Cuando el sistema es de segundo orden la representación se adecúa fácilmente en un plano. Para sistemas de orden tres es posible la proyección de un trazo tridimensional, aunque en la mayoría de los casos es preferible la representación en gráficas separadas de cada una de las variables.

Como herramienta de representación gráfica utilizamos las posibilidades del paquete MATLAB. Éste es, en realidad, un interprete de un lengúaje de alto nivel que tiene como elemento básico las matrices de dos dimensiones [Matl 92]. En sus ultimas versiones posee unas excelentes capacidades gráficas en dos y tres dimensiones que lo hacen muy adecúado a nuestros propósitos. Posee además una gran facilidad de manejo y sobre todo de ampliación que hacen posible el diseño de funciones propias.

Por otro lado, por la capacidad de representación matricial, es propicio a ser utilizado en la fase de diseño y comprobación inicial de los algoritmos [Demu 93] [Grac 92]. Nos permite realizar ejemplos a pequeña escala sin necesidad de codificar largos programas. También utilizamos sus matrices para el almacenamiento de los resultados de manera accesible para su posterior comparación.

Este paquete está disponible para numerosas plataformas entre ellas las utilizada por nosotros. De esta manera se facilitaba el proceso de cálculo y estudio simultáneo al tratarse de un sistema multiproceso.

7.2 Codificación

En esta sección presentaremos las características fundamentales de las estructuras de datos y algoritmos utilizados para la implementación de los distintos métodos. La descripción aquí realizada se hará a alto nivel y en un seudocódigo estructurado. En el apéndice A se lista el código fuente completo de los principales programas.

7.2.1 Programación Dinámica

7.2.1.1 Estructura de Datos

En la PD la principal estructura de datos es la rejilla de discretización, es decir, una matriz *n*-dimensional. Cada uno de los elementos representa a un punto del espacio de estado si, como en nuestro caso, el tiempo no es la variable de etapa. En cada uno de estos puntos de la rejilla se han de almacenar la función de costo mínimo y los comandos óptimos. También, para facilitar el segundo barrido, almacenamos el índice del estado de la etapa siguiente que se alcanza si aplicamos el comando óptimo. Por utilizarse una discretización uniforme las coordenadas del estado que se representa en cada punto se determinan por su posición dentro de la estructura.

En C definimos una estructura `TNodo` en la que se definen los campos para los distintos datos a almacenar en un elemento. El tamaño total de la rejilla viene determinado por el número de puntos que deseemos en cada dimensión. La memoria requerida se solicita en tiempo de ejecución según las características del ensayo, utilizándose la mínima necesaria. Como los nodos quedan como un array monodimensional ha de ser el algoritmo el que lo organice como matriz *n*-dimensional. De esta forma, aunque se complica el código, se adecúa a un recorrido mediante punteros, que lo hacen muy eficiente.

Para facilitar el primer barrido de la PD la dimensión que avanza más lentamente ha de ser la de la variable de estado elegida como variable de etapa. Es decir, todos los puntos de una etapa estarán almacenados consecutivamente. Dentro de cada etapa la organización de las otras dimensiones no es trascendental.

7.2.1.2 Algoritmos

El cálculo de la PD se realiza mediante dos programas distintos que corresponden a cada uno de los barridos de ésta. Una vez calculados los datos con el primer programa, y almacenados en un fichero, nos valen para distintas ejecuciones del segundo.

La estructura del programa principal que realiza el primer barrido de la PD queda de la siguiente manera:

```
Solicitar parámetros del ensayo
Crear estructura de rejilla
Realizar el primer paso de la PD
Gúardar los resultados en fichero
```

La creación de la estructura de la rejilla, como comentamos en el apartado anterior, consiste principalmente en solicitar la memoria necesaria. También es preciso inicializar convenientemente algunos puntos de ésta: a los puntos finales se les asigna una función

de costo mínimo igúal a cero y a los puntos inalcanzables un costo infinito. También se determinan en este momento que puntos formarán parte de la frontera entre las regiones de evolución de la variable de etapa. En caso de interdependencia de las regiones también será necesario inicializar el coste de alguno de estos puntos de la frontera.

Para la realización del primer paso de la PD se ha de comenzar por la región que tenga determinados todos sus puntos finales y se ejecuta el algoritmo presentado en la sección 4.1.3.1. Como mencionamos anteriormente, para cada punto no sólo se almacenarán el costo mínimo y los comandos óptimos, sino también el índice del punto de la etapa siguiente al que se llega con dichos comandos óptimos. Se sigue una convención especial para marcar los puntos finales e inalcanzables ya que, por distinto motivo, estos no tienen estado siguiente.

El almacenamiento de los resultados en ficheros es sencillo. En primer lugar se gúardan los parámetros que determinan la forma y tamaño de la rejilla. A continúación se copia directamente la zona de memoria que almacena a ésta. Esto es posible porque en los datos no existen referencias a direcciones absolutas de memoria, es decir, los índices almacenados corresponden a los de la matriz *n*-dimensional y no a la dirección de memoria en donde se encuentran los elementos referenciados.

La estructura del programa que realiza el segundo barrido de la PD es la siguiente:

```
Leer rejilla de fichero
Pedir puntos iniciales
Realizar el segundo barrido
Almacenar las trayectorias calculadas
```

Para recuperar los datos de la rejilla se pueden copiar directamente a memoria, una vez solicitada ésta. Los puntos iniciales solicitados corresponden a aquellos para los cúales deseamos obtener su trayectoria óptima. Se comprueba que estos caigan dentro de la zona de espacio de estado estudiada y se aproximan al punto de la rejilla más próximo si no coinciden con ninguno de ellos.

En la realización del segundo barrido, una vez sitúado en el nodo correspondiente al estado inicial, se va pasando al estado de la etapa siguiente apuntado por los índices almacenados en dicho nodo. De esta manera se seguirá hasta llegar a un punto final o uno de retorno. En este segundo caso debemos continúar pero estaremos en otra región, con lo cúal cambiará la forma de avance de la variable de etapa. Vamos almacenando en un array monodimensional los estados por los que vamos pasando y los comandos óptimos que figuran en dichos puntos. Esto constituye la información de la trayectoria óptima.

En este mismo procedimiento se ha de tener en cuenta la existencia de simetría en el sistema. Si el punto inicial corresponde a la región complementaria a la que hemos tratado, debemos sitúarnos en el punto complementario a éste y calcular a partir de ahí la trayectoria. A los datos así obtenidos se les ha de aplicar la transformación de simetría antes de almacenarlos en el array de trayectoria.

Finalmente las trayectorias para todos los puntos iniciales se salvan en un fichero con el formato preciso para ser leído por MATLAB, es decir, como una matriz bidimensional de números reales.

7.2.2 Métodos de Redes Neuronales

En los dos métodos basados en redes neuronales presentados en este trabajo las estructuras de datos y de control son muy similares. Por esa razón presentamos su implementación conjuntamente.

7.2.2.1 Estructura de Datos

La estructura de datos, comparada con la de la PD, es mucho más compleja. Las características, en orden de importancia, que exigimos en el momento de diseñarla son: eficiencia en el cálculo, flexibilidad y uso de mínima memoria. La flexibilidad significa la posibilidad de elegir una estructura diferente para cada una de las redes de la cadena, tanto en número de capas como en neuronas en cada capa. Esta estructura ha de poderse fijar en tiempo de ejecución y por lo tanto la memoria ha de solicitarse dinámicamente para utilizar la estrictamente necesaria.

Debido a esta flexibilidad se hace un diseño estructurado, es decir, en cada nivel sólo se conoce lo estrictamente necesario de los niveles inferiores. Existe una estructura de tipo `TRed` en la que se almacenan distintos parámetros del sistema y, en cúanto a la estructura, únicamente el número de etapas y un puntero a un array del tipo de datos inferior. Este array tiene N elementos de tipo `TUnidad`, en el que se gúarda la información específica de cada red. Si seguimos con la estructuración, de ésta colgarían estructuras que representan a las capas, pero esto sería poco eficiente para el cálculo, ya que las capas se encuentran muy interrelacionadas. Optamos por colgar de `TUnidad` directamente arrays con los datos organizados por tipos: pesos, entradas, valores intermedios, etc. También se consideran los datos necesarios para el entrenamiento: pesos del paso anterior de entrenamiento, parámetros δ, vector gradiente, etc. Se almacenan los necesarios para todas las variantes de la BP ya que éstas se pueden seleccionar en tiempo de ejecución.

Para determinar la estructura dentro de estos arrays es necesario conocer el número de neuronas en cada capa. Un campo de la estructura `TUnidad` indica el número de

capas de la red (L^i) mientras que en un array se gúarda ordenadamente el número de neuronas por capa. Así mismo, como el rango y el tipo de función de las neuronas de salida puede ser distinto, también se almacena esta información.

De toda la información que se considera en una red, mucha de ella es temporal. Por ejemplo, el valor de las salidas, los valores intermedios z, el valor de los parámetros δ, etc., pueden calcularse a partir de las restantes dada una entrada particular. Para facilitar el manejo de la información fundamental en el momento de almacenarla en ficheros, agrupamos la de todas las redes consecutivamente, es decir, en una única petición de memoria. Posteriormente, bastará con hacer apuntar los distintos campos de la estructura `TUnidad` a la posición de memoria correcta. Para poder mover este bloque fácilmente en su primera posición se almacena el número de bytes que lo componen. La estructura real entonces es la mostrada en la figura 7.1.

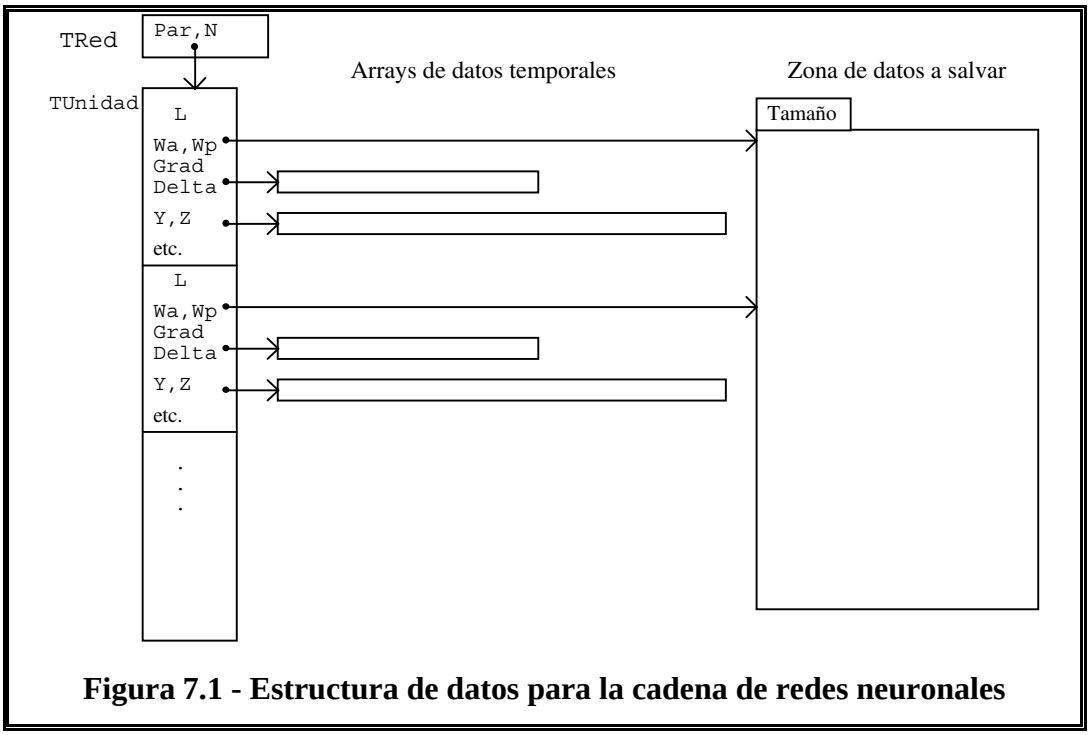

Figura 7.1 - Estructura de datos para la cadena de redes neuronales

7.2.2.2 Algoritmos

En este caso se combinan en un único programa tanto el cálculo como el proceso de entrenamiento de la cadena. Esto es así porque el cálculo de la red es necesario en el proceso de corrección mediante la BP. Como el número de entrenamientos no está determinado a priori, en una ejecución se puede o bien comenzar con una nueva red, o bien continúar con el entrenamiento de una cadena ya entrenada. Además, para estudiar la evolución, es necesario obtener las trayectorias para ciertos puntos. Por estas consideraciones se ha optado por una estructura de programa principal en forma de menú. Éste quedará:

```
Si se indica fichero
   Leer datos de fichero
Caso contrario
   Crear estructuras
Mientras no se elija salir
   Presentar menú
   Según la opción elegida
      Modificar estructura
      Calcular Trayectorias
      Entrenar
      Salvar estado de la cadena
      Modificar Parámetros
      Salvar datos
Liberar memoria y salir
```

En el proceso de creación de la estructura se comienza preguntando por el número de etapas. A continúación se pregunta por el número de capas típico de las redes. Este número se aplica a todas las redes y seguidamente se da la posibilidad de modificarlo en una red concreta. De la misma manera se opera con el número de neuronas en cada capa, se pregunta el valor típico y se da la posibilidad de modificar el caso particular. Una vez creada la estructura se pregunta por la función de activación y escalado de las neuronas de salida, pudiendo hacerse especificaciones por red. Finalmente se hacen las cuestiones referentes a la inicialización de los pesos. Ésta puede ser aleatoria o leída de un fichero MATLAB, conveniente si procede de un entrenamiento individúalizado. Es también en este momento cúando se cargan los datos referentes a la trayectoria pesada o a la zona prohibida si se desean utilizar.

En la opción de cálculo de las trayectorias, de forma similar a como ocurría en la PD, se han de indicar los puntos iniciales para los cúales deseamos dicho cálculo. Como en este caso el entrenamiento se realiza para una región A^0 existe la posibilidad de calcular automáticamente las trayectorias para los puntos extremos de dicha región, lo cúal nos permite observar el grado de optimización global. Una vez calculadas las trayectorias son almacenadas en el fichero indicado con el formato MATLAB y con estructura idéntica a la utilizada en la PD (estructura que veremos en el apartado 7.2.3.1).

El entrenamiento puede realizarse según dos opciones. La primera está pensada para un entrenamiento breve realizado interactivamente, ya que no se gúarda información alguna del proceso. La segunda, en cambio, está pensada para largos procesos de entrenamiento. Además del número total de entrenamientos que queremos alcanzar se han de indicar intervalos parciales en los cúales se almacenan automáticamente

estadísticas acerca del entrenamiento. Los principales datos almacenados son: costo promedio de las trayectorias para ciertos puntos iniciales, módulo promedio del vector gradiente, paso de corrección aplicado (útil en el caso de búsqueda de la dirección del gradiente), etc. Como en este caso los periodos de tiempo son mucho mayores se evita la salida por pantalla para posibilitar que el proceso sea puesto en *background* sin que se detenga su ejecución. Cúando se alcanzan los entrenamientos indicados toda esta información se gúarda automáticamente en formato MATLAB y también se salva la cadena en estado actúal.

Se salva el estado de la cadena para que pueda ser tratado en una ejecución posterior por el mismo programa. Este procedimiento se realiza comenzando por la estructura `TRed`. A continúación, del array de estructuras `TUnidad`, se almacena sólo la información referente a la estructura de cada red, es decir, número de capas y número de neuronas por capa. Finalmente se almacena el bloque continuo con toda la información esencial que, como dijimos en el apartado anterior, está consecutiva en memoria.

Existen opciones para modificar los numerosos parámetros del proceso. Estos parámetros se dividen en los que corresponden al método de entrenamiento y los que son independientes de la variante de la BP elegida. En los parámetros de entrenamiento podemos combinar las distintas técnicas de aceleración de la BP: gradiente conjugado en sus distintas variantes; búsqueda en la dirección del gradiente o paso fijo; escalado del gradiente según la capa; promedio de varios gradientes antes de corregir; etc. Los parámetros más generales consisten, fundamentalmente, en la región inicial de entrenamiento. Se solicitan valores inferiores y superiores para cada una de las variables de estado. Con estos valores se definen intervalos, dando así lugar a un cubo n-dimensional.

En la opción de salvar datos se posibilita almacenar, en un fichero para MATLAB, algunas de las variables internas de las redes como son los pesos, variables z, variables δ, etc. Esto permite su estudio y representación de una forma cómoda para detectar así patologías en la red.

7.2.3 Estudio en MATLAB

7.2.3.1 Estructuras de Datos

Como dijimos previamente, será en MATLAB donde haremos el tratamiento de los resultados. Como indicamos, la única estructura de datos en este paquete es la matriz bidimensional, aunque se distinguen casos especiales como vectores y escalares. A pesar

de esta aparente limitación, siguiendo unos convenios las matrices se adecúan para almacenar gran variedad de datos.

En todos nuestros casos el resultado principal es la trayectoria. Hemos definido un formato común en el que se gúardan las trayectorias de todos los métodos. Para almacenarlas en matrices cada fila corresponde a una de las etapas de la trayectoria. En las primeras columnas se almacenarán los estados en dicha etapa. A continúación se almacenarán los comandos y el intervalo de tiempo aplicados en esa etapa. En la última columna de cada fila se almacena el costo acumulado desde el principio de la trayectoria hasta esa etapa, o el costo total si se trata de la última. De esta manera se puede almacenar una trayectoria completa por matriz.

Deseábamos un formato más compacto que permitiera gúardar información sobre el sistema que generó la trayectoria y, además, tener la posibilidad de almacenar varias trayectorias en una única matriz. Para conseguir esto añadimos tres filas, delante de las de la trayectoria, en las que se colocan los parámetros más destacados. El primer elemento de esas nuevas filas indica el número de estados de las trayectorias. De esta manera, cúando estén agrupadas varias en una matriz, se puede determinar en que fila comienza la siguiente trayectoria. Además, al ser una indicación relativa, se pueden mover las trayectorias, junto con sus filas de parámetros, a otras matrices sin que se pierda la estructura.

En el caso de las redes neuronales también queremos una estructura de datos con información sobre el proceso de entrenamiento. Para ello en cada fila se almacena información correspondiente a un determinado paso de entrenamiento. El primer elemento de cada fila indica el número de entrenamiento del que se trata, mientras que en las restantes columnas se apuntan los otros parámetros como son: costo de la trayectoria probada, magnitud del gradiente, paso de entrenamiento utilizado, etc. De esta manera es posible representar la evolución de dichos parámetros con el entrenamiento.

Para salvar variables de la red, como son los pesos, gradiente, variables δ, etc. se crean matrices en las que cada columna corresponde a una red. En las filas se va colocando el valor de las variables partiendo de la primera neurona de la primera capa, siguiendo con todas las neuronas de la primera capa y así sucesivamente hasta terminar con todas las capas. Este formato, entonces, es sólo válido para el caso en que todas las redes sean igúales, ya que todas las columnas han de tener la misma longitud. En caso de no ser así habría que añadir una primera fila en que se indique cúantos datos hay en esa columna y elegir una matriz de tamaño suficiente para que quepa la columna más grande.

7.2.3.2 Nuevas Funciones

Para MATLAB se han diseñado una serie de nuevas funciones para tratar con los tipos de datos mencionados anteriormente.

De especial importancia son las que manejan las matrices de trayectorias. Se han escrito numerosas funciones que: devuelven el número de trayectorias en la matriz, nos permiten extraer una del conjunto, extraer información sobre el punto inicial y final y el costo total, etc. Entre las más utilizadas están las de representación de las trayectorias. Esta representación, como mencionamos al principio del capítulo, se realiza en dos dimensiones. La forma más correcta es representar la evolución de las variables de estado y comandos frente al tiempo. Con la aparición de intervalos de tiempo negativos en el segundo método esta representación se hacia engorrosa, ya que las trayectorias y comando volvían hacia atrás. En esos casos se optó por representar las distintas magnitudes, incluso el intervalo de tiempo, frente a la etapa.

Para las estadísticas de los entrenamientos las funciones consistían en representar las magnitudes frente al número de entrenamientos.

Con las matrices que contenían los pesos de la cadena se hacia un tratamiento exhaustivo para determinar la sensibilidad de las capas. También se determinaban los valores singulares por capas. Representando estos resultados en distintos pasos de entrenamiento se detectaban sitúaciones anómalas.

Principales Aportaciones, Conclusiones y Líneas Abiertas

A continuación se resumen las principales aportaciones de este trabajo:

- Se ha formulado y demostrado un teorema que define las condiciones suficientes para poder encontrar, en un problema de PD general, una variable de etapa distinta del tiempo. De esta manera es posible abordar índices de tiempo mínimo.
- Se ha formulado y demostrado un teorema que define condiciones suficientes y necesarias para reducir, al menos a la mitad, en un problema de PD general, la complejidad espacial y temporal de ésta.
- Se han aplicado estos teoremas, junto con otras técnicas, a tres sistemas no lineales elegidos con diferentes características y complejidades. De esta aplicación podemos concluir que, aunque con las reducciones conseguidas es posible tratar sistemas de orden superior al segundo, la optimalidad de las soluciones está condicionada a la disponibilidad de memoria.
- Se ha diseñado un primer método basado en RN alternativo a la PD en la solución de problemas de control óptimo generales para puntos de una

determinada región inicial. Este método consiste en una cadena de redes que producen el comando e intervalo de tiempo a partir del estado del sistema. Como ha de ser el modelo dinámico del sistema el que produzca el estado siguiente que se alcanza, no es posible utilizar discretización BPF.

- Se han desarrollado las fórmulas, basadas en el algoritmo de la BP, para entrenar las redes de este primer método de manera que los comandos producidos sean los óptimos de acuerdo a la función de costo planteada. Esta función de costo así como las ecuaciones del sistema pueden ser generales.
- Se ha aplicado este primer método de RN a los problemas antes citados. Se observa que la optimalidad de los resultados dependen del tamaño de la región inicial entrenada. Se obtienen resultados mejores que la PD para regiones de tamaño moderado, pero si se parte de puntos exteriores a éstas las trayectorias generadas no conducen a los puntos finales deseados.
- Se ha diseñado un segundo método basado en RN para solucionar también problemas de control óptimo generales para puntos de una región inicial. Este método consiste en una cadena de redes que producen, en este caso, el estado siguiente a partir del estado actual del sistema. Como ha de ser el modelo dinámico del sistema el que calcule los comandos y el intervalo de tiempo necesario para la transición, es posible utilizar discretización BPF.
- Se han desarrollado las fórmulas, basadas en el algoritmo de la BP, para entrenar las redes de este segundo método de manera que los estados producidos formen la trayectoria óptima de acuerdo a la función de costo planteada. Tanto la función de costo como las ecuaciones del sistema pueden ser generales.
- Se han aplicado distintas técnicas de aceleración y mejora del algoritmo de entrenamiento para conseguir la convergencia en el proceso de este segundo método de RN.
- Se ha aplicado este segundo método de RN a los problemas antes citados. Al contrario que en el primer método, para puntos exteriores de la región inicial, las trayectorias generadas regresan a la óptima, alcanzándose el punto final.
- Al igual que ocurria en la PD, se ha demostrado que el uso de la formulación BPF produce mejores resultados que la discretización clásica. Esto significa una virtud del segundo método de RN en comparación al primero.

A la vista de los resultados de este trabajo surgen numerosas opciones que parece interesante considerar. Entre las más importantes son:

- Utilización y combinación de otros algoritmos de entrenamiento para solucionar las dificultades de convergencia del segundo método. Entre estos pueden estar los algoritmos genéticos, el método tabú, etc.
- Combinación de varias cadenas de redes para cubrir regiones más amplias del espacio de estados y así paliar el problema del primer método de redes neuronales.
- Cambio en la estructura de utilización de las redes neuronales, como por ejemplo asignar a cada red una región del espacio de estados.
- Combinar estos métodos de redes con otras redes dinámicas que identifiquen los sistemas, produciendo así un control adaptativo.

Apéndice A

En este apéndice se lista el código fuente de los principales programas tanto de simulación de los métodos como de tratamiento de los resultados. Comenzamos con los correspondientes a la PD. Estos programas son diferentes para cada sistema, por lo cual los separamos en apartados distintos. En cuanto a los de RN, existe una gran parte común a ambos métodos y a todos los sistemas. Por eso, en la segunda sección, se comienza con el código común y, en los apartados por sistemas, se listan los ficheros cabecera que determinan el método y el tipo de discretización utilizada. Hasta aquí todo el código es en lenguaje C, por lo que los ficheros tienen extensión `c` y `h`. Finalmente, en la última sección de este apéndice, se agrupan las funciones realizadas para el paquete MATLAB que, como mencionamos en el capítulo 7, ha sido utilizado para tratar y visualizar los resultados. Los ficheros en este caso tiene extensión `m`.

A.1 Programación Dinámica

A.1.0.1 tdatos.h

```
typedef double Real;
typedef unsigned short SinSigno;
typedef unsigned long LargoSS;
```

```
typedef unsigned char TLogico;
#define FALSO 0
#define VERDAD 1


typedef struct {
     long tipo;
     long NumFilas;
     long NumColum;
     long HayImag;
     long TamNom;
}  CabezaMat;

extern double drand48();
```

A.1.1 Primer Sistema

A.1.1.1 dinamica.h

```
/* definicion de tipos */
#define  Real double
typedef unsigned long SinSigno;
#define MAX_INT  0xFFFFFFFF

typedef struct
   {
      Real I,U,Tau;
      SinSigno jsig;
    } TipoDato;

typedef struct
   {
      SinSigno N1menos,N1Mas,N2;
      Real      d1,d2;
      Real  A,B,C,D;
      Real CotaCom,FacT,FacCom,FacX1,FacX2;
   } TParametros;
#define NUMPAR 14

typedef struct {
     long tipo;
     long NumFilas;
     long NumColum;
     long HayImag;
     long TamNom;
}  CabezaMat;

#define MAXPUNDEF  1000

typedef struct
{
   SinSigno MaxPun,iini,jini;
   int SigPlano;
}  Condiciones;

typedef struct
{
   Real x1,x2,consig,tau,costo;
} PuntoTrayec ;  /* datos en cada punto de la trayectoria */

#define ROUND(x)  ((fabs((x)-floor(x))>.5)?ceil(x):floor(x))
#define NUMMAXT 20


#include "inter.h"
```

A.1.1.2 inter.h

```
#include <math.h>

#define CalculaSumaU(p,x1k,x2k,x1k1,x2k1)  (1/p.C*((x2k1+x2k)/(x1k1-
x1k)*(x2k1-x2k)-p.A*(x2k1+x2k)-p.D*(x1k1+x1k)-
p.B*(x1k1*x1k1*x2k1+x1k*x1k*x2k)))
#define CalculaTau(p,x1k,x2k,x1k1,x2k1)  (2*(x1k1-x1k)/(x2k1+x2k))
#define IndiceL(p,x1k,x2k,u,tau)
((Real)p.FacT*(Real)fabs(tau)+(Real)p.FacCom*(Real)fabs(u)+(Real)p.FacX1*(Re
al)fabs(x1k)+(Real)p.FacX2*(Real)fabs(x2k))
```

A.1.1.3 calcula.c

```
#include "dinamica.h"
#include <stdio.h>
#include <stdlib.h>
#ifdef __unix

   int ObtenerParametros();
   void  Inicializa();
   int Salva();
   int SalvaMat();
   int Muestra();
   void Procesa();

/* funciones que dependen del problema */
   void Procesa();
#else
void ObtenerParametros(TParametros *);
void  Inicializa(TParametros param,TipoDato *Puntos);
int Salva(TParametros *param,TipoDato *puntos,char *nombre);

/* funciones que dependen del problema */
Real CalculaSumaU(Real x1act,Real x2act,Real x1pru,Real x2pru);
Real CalculaTau(Real x1act,Real x2act,Real x1pru,Real x2pru);
Real IndiceL(Real x1act,Real x2act,Real u,Real tau);
```

```
#endif

main(argc,argv)
int argc;
char *argv[];
{
   TParametros para;
   TipoDato  *puntos;
   SinSigno menreq;
   char NomFich[4][13];

   if(PreparaFich(argc,argv,NomFich))
   {  /* las opciones son correctas */
      if(!ObtenerParametros(&para,NomFich[3]))
      {
         printf("\nERROR.No se pueden obtener parametros\n");
         return 0;
      }
      menreq=(para.N1menos+para.N1Mas+1)*(para.N2+1)*sizeof(TipoDato);

      if (puntos=malloc(menreq))
         {  /* se ha conseguido la memoria para los datos */
            Inicializa(para,puntos);

            Procesa(para,puntos);

            if (!Salva(para,puntos,NomFich[0]))
            {
              printf("\n ERROR: No puedo salvar los datos binarios");
              printf(" Fichero: %s\n",NomFich[0]);
              return 1;
            }
            if (NomFich[1][0] && !SalvaMat(para,puntos,NomFich[1]))
            {
              printf("\n ERROR: No puedo salvar los datos para MATLAB ");
              printf(" Fichero: %s\n",NomFich[1]);
              return 1;
            }
            if (NomFich[2][0] && !Muestra(para,puntos,NomFich[2]))
            {
              printf("\n ERROR: No puedo salvar los datos en ASCII.");
              printf(" Fichero: %s\n",NomFich[2]);
              return 1;
            }

            free(puntos);
            return 0; /* convenio 0 cuando terminacion correcta */
         }
         else
            printf("\n ERROR.No conseguida la memoria para datos\n");
   }
   else
      printf("\n ERROR ** opciones incorectas \n");
return 1; /* convenio de sacar 1 cuando ha existido error */
}
```

A.1.1.4 crea.c

```
#include "dinamica.h"
#include <stdio.h>
#include <stdlib.h>


#ifdef     unix
   int Carga();
   int GuardaTraMat();
   int GuardaTraSal();
   int PreparaFich();
   unsigned EntraIni();
#else

#endif


SinSigno Reconstruye(param,puntos,Cini,trayec)
TParametros param;
TipoDato *puntos;
Condiciones Cini;
PuntoTrayec *trayec;
{
#define   DIR(i,j)   puntos+i*(param.N2+1)+j
#define  X1(i)  ((Real)(i)-(Real)param.N1menos)*param.d1
#define  X2(j)  (-(Real)(j))*param.d2

   TipoDato *PtDatos;
   PuntoTrayec *PuntoAct;
   SinSigno iact,jact,NumPunt;
   int Semiplano;
   Real CostoAct;

   Semiplano=(Cini.SigPlano>0)?-1:1; /* -1 para semiplano superior */
   CostoAct=0.0;
   for(iact=Cini.iini,jact=Cini.jini,NumPunt=0,PuntoAct=trayec;
       iact && (iact!=param.N1menos || jact) && (NumPunt<Cini.MaxPun-1);
       iact--,jact=PtDatos->jsig,NumPunt++,PuntoAct++)
      {
         if (!jact && iact<param.N1menos)
            {
               iact=2*param.N1menos-iact;
               if (iact>(param.N1menos+param.N1Mas))
                     return NumPunt; /* nos salimos de la regilla */
               Semiplano= -Semiplano; /* cambiamos de semiplano */
            }
         PtDatos=DIR(iact,jact);
         if (PtDatos->jsig==MAX_INT)
            break; /* Punto que no tiene continuacion */

         /* Copiamos datos en array de trayectoria */
         PuntoAct->x1=Semiplano*X1(iact);  /* Signo afectado por  */
         PuntoAct->x2=Semiplano*X2(jact);  /* el semiplano */
         PuntoAct->consig=Semiplano*(PtDatos->U);
```

```
            PuntoAct->tau=PtDatos->Tau;
            PuntoAct->costo=CostoAct;
            CostoAct=(Real)IndiceL(param,PuntoAct->x1,PuntoAct->x2,PuntoAct-
>consig,PuntoAct->tau)+CostoAct;
      } /* del for */
      /* llegamos al (0,0) o demasiados puntos */
      /* almacenamos ultimo punto */
      PtDatos=DIR(iact,jact);
      PuntoAct->x1=Semiplano*X1(iact);
      PuntoAct->x2=Semiplano*X2(jact);
      PuntoAct->consig=Semiplano*(PtDatos->U);
      PuntoAct->tau=PtDatos->Tau;
      PuntoAct->costo=CostoAct;  /* costo total de la trayectoria */

      return NumPunt+1;
} /* de Reconstruye */


main(argc,argv)
int argc;
char *argv[];
{
      TParametros param;
      TipoDato *puntos;
      unsigned NumTrayec,k;
      Condiciones CondIni[NUMMAXT];
      SinSigno PunTrayec[NUMMAXT];
      PuntoTrayec *trayectoria[NUMMAXT];
      char NomFich[4][13];

      if(PreparaFich(argc,argv,NomFich))
      {
         if(Carga(&param,&puntos,NomFich[0]) &&
            (NumTrayec=EntraIni(param,CondIni,NomFich[3])))
         {  /* optenidos datos correctamente */
            for (k=0;k<NumTrayec;k++)
            {
               if(trayectoria[k]=malloc(CondIni[k].MaxPun*sizeof(PuntoTrayec)))

PunTrayec[k]=Reconstruye(param,puntos,CondIni[k],trayectoria[k]);
               else
                  return 0; /* no hay memoria de datos */
            }
            if (NomFich[1][0] &&
               !GuardaTraMat(trayectoria,PunTrayec,NumTrayec,NomFich[1]))
            {
              printf("\n ERROR: No puedo salvar los datos para MATLAB.");
              printf(" Fichero: %s\n",NomFich[1]);
              return 1;
            }
            if(NomFich[2][0] &&
               !GuardaTraSal(param,trayectoria,PunTrayec,NumTrayec,NomFich[2]))
            {
              printf("\n ERROR: No puedo salvar los datos en ASCII.");
              printf(" Fichero: %s\n",NomFich[2]);
              return 1;
            }

            for (k=0;k<NumTrayec;k++)
               free(trayectoria[k]);
            free(puntos);
            return 0; /* cuando terminacion correcta */
         }
         else
            printf("\n ERROR:imposible obtener datos o condiciones
iniciales\n");
      }
      else
         printf("\n ERROR: Opciones incorrectas \n");
      return 1;  /* 1 cundo ha habido error */
}  /* de main */
```

A.1.1.5 depende.c

```
#include "dinamica.h"
#include <math.h>


TipoDato Dato0={ 0.0, 0.0, 0.0, 0},
         DatoImpo={ HUGE_VAL,0.0,0.0, MAX_INT};
void  Inicializa(param,Puntos)
TParametros param;
TipoDato *Puntos;
{ /* Pone los valores de frontera:
            - a 'inalcanzable' el extremo izquierdo.
            - a 0 el eje x1 negativo.    */

   TipoDato *PtAct;
   SinSigno i;

   /* Para el extremo izquierdo */
   for(i=0,PtAct=Puntos;i<=param.N2;i++,PtAct++)
         *PtAct=DatoImpo;

   /* para el eje X1 no positivo */
   for(i=0,PtAct=Puntos;i<=param.N1menos;i++,PtAct+=(param.N2+1))
         *PtAct=Dato0;
}
```

A.1.1.6 detalle.c

```
#include "dinamica.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LARGOX 15
#define LARGOY 6
```

```
int Carga(param,DatosPt,NomFich)
TParametros *param;
TipoDato **DatosPt;
char *NomFich;
{
   FILE *FEnt;
   SinSigno NumDatos;

   if(FEnt=fopen(NomFich,"r"))
   {
      /* leer los  parametros */
      if(1!=fread((char *)param,sizeof(TParametros),1,FEnt))
         return 0;  /* error por falta de datos*/

      NumDatos=(param->N1menos+param->N1Mas+1)*(param->N2+1);
     /* obtenemos memoria para datos */
      if(*DatosPt=malloc(NumDatos*sizeof(TipoDato)))
      {
         /* leer datos de los puntos ya que tenemos memoria */
         if(NumDatos!=fread((char
*)*DatosPt,sizeof(TipoDato),(size_t)NumDatos,FEnt))
         {
            free(*DatosPt);
            return 0;
         }
      }
      else return 0; /* falta de memoria */

      return 1; /* terminacion correcta */
   }
   else
      return 0; /* error no se puede abrir fichero */
} /* de Carga */



PreparaFich(numarg,argu,NomF)
int numarg;
char *argu[],NomF[4][13];
{
   char Nombre[9];
   int i,pos;
   NomF[0][0]='\0';  /* inicializa los nombres a vacio */
   NomF[1][0]='\0';
   NomF[2][0]='\0';
   NomF[3][0]='\0';
   Nombre[0]='\0';

   /* Obtener nombre base */
   for(i=1;!*Nombre && i<numarg;i++)
      if (*argu[i]!='-')
      {
         char *aux;
         aux=strchr(argu[i],'.');
         pos=(aux)? (aux-argu[i]):10;
         strncpy(Nombre,argu[i],(pos>8?8:pos));
         Nombre[8]='\0';
      };
   if (*Nombre)
   {  /* existe nombre base para los ficheros */
      strcpy(NomF[0],Nombre);
      strcat(NomF[0],".dat");  /* crea fichero datos */
      for(i=1;i<numarg;i++)
        /* busca las opciones */
        if (argu[i][0]=='-')
           switch (argu[i][1]) {
           case 'm':
              NomF[1][0]='\0';
              if (strlen(argu[i])>2)
              {
                 strncpy(NomF[1],argu[i]+2,12);
                 NomF[1][12]='\0';
              }
              else {
                 strcpy(NomF[1],Nombre);
                 strcat(NomF[1],".mat");
                 };
              break;
           case 's':
              NomF[2][0]='\0';
              if (strlen(argu[i])>2)
              {
                 strncpy(NomF[2],argu[i]+2,12);
                 NomF[2][12]='\0';
              }
              else {
                 strcpy(NomF[2],Nombre);
                 strcat(NomF[2],".slc");
                 };
              break;
           case 'e':
              NomF[3][0]='\0';
              if (strlen(argu[i])>2)
              {
                 strncpy(NomF[3],argu[i]+2,12);
                 NomF[3][12]='\0';
              }
              else {
                 strcpy(NomF[3],Nombre);
                 strcat(NomF[3],".enc");
                 };
           }; /* del switch el if y el for */
      return 1; /* preparados los nombres */
   } /* de if existe nombre */
   return 0; /* no se encontro nombre base */
}  /* de PreparaParam */

void Presenta(par,punt,Tc,Tf,px,py)
TParametros par;
TipoDato *punt;
unsigned int Tc,Tf,px,py;
{
   unsigned int cini,cfin,lini,lfin,i,j;
```

```
   cini=(px<= Tc/2)?0:px-Tc/2;
   cfin=cini+Tc-1;

   lini=(py<= Tf/2)?0:py-Tf/2;
   lfin=lini+Tf-1;


   printf("\n      ");
   for(i=cini;i<=cfin;i++)
      printf("   %5.2lf %3u   ",(i*par.d1-par.N1menos*par.d1),i);
   printf("\n");

   for(j=lini;j<=lfin;j++)
   {
      printf("\n");
      printf("%5.2lf ",-par.d2*j);
      for(i=cini;i<=cfin;i++)
         printf(" %11.4g=U ",(punt+i*(par.N2+1)+j)->U);
      printf("\n %3u  ",j);
      for(i=cini;i<=cfin;i++)
         printf(" %11.4g=T ",(punt+i*(par.N2+1)+j)->Tau);
      printf("\n      ");
      for(i=cini;i<=cfin;i++)
         printf(" %11.4g=I ",(punt+i*(par.N2+1)+j)->I);
      printf("\n      ");
      for(i=cini;i<=cfin;i++)
         printf(" %10lu=js ",(punt+i*(par.N2+1)+j)->jsig);
      printf("\n");
   }
}

main(argc,argv)
int argc;
char *argv[];
{
   TParametros param;
   TipoDato *puntos;
   char NomFich[4][13];
   char *respuesta;
   unsigned int lineas,columnas,TotCol,TotFil,fele,cele;
   Real px1,px2;

   if(PreparaFich(argc,argv,NomFich))
   {
      if(Carga(&param,&puntos,NomFich[0]))
      {  /* optenidos datos correctamente */

         respuesta=getenv("LINES");
         sscanf(respuesta,"%u",&lineas);

         respuesta=getenv("COLUMNS");
         sscanf(respuesta,"%u",&columnas);

         TotCol=(columnas-6)/LARGOX;
         TotFil=(lineas-1)/LARGOY;

         printf("l:%u  c:%u\n\n",lineas,columnas);

         printf(" Entra coordenadas a ver.Coord x1:");
         scanf("%lf",&px1);
         printf("Coord x2:");
         scanf("%lf",&px2);

         while(px2<param.d2)
         {
            cele=(unsigned int)(ROUND(px1/param.d1)+param.N1menos);
            fele=(unsigned int)(ROUND(fabs(px2/param.d2)));

            if ((cele<param.N1menos+param.N1Mas) && fele<param.N2)
              Presenta(param,puntos,TotCol,TotFil,cele,fele);

            printf(" Entra coordenadas a ver.Coord x1:");
            scanf("%lf",&px1);
            printf("Coord x2:");
            scanf("%lf",&px2);

         }




      }
      else
         printf("\n ERROR:imposible obtener datos o condiciones
iniciales\n");
   }
   else
      printf("\n ERROR: Opciones incorrectas \n");
   return 1;  /* 1 cundo ha habido error */
}  /* de main */
```

A.1.1.7 iocrea.c

```
#include "dinamica.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int Carga(param,DatosPt,NomFich)
TParametros *param;
TipoDato **DatosPt;
char *NomFich;
{
   FILE *FEnt;
   SinSigno NumDatos;

   if(FEnt=fopen(NomFich,"r"))
   {
```

```
      /* leer los  parametros */
      if(1!=fread((char *)param,sizeof(TParametros),1,FEnt))
         return 0;  /* error por falta de datos*/

      NumDatos=(param->N1menos+param->N1Mas+1)*(param->N2+1);
     /* obtenemos memoria para datos */
      if(*DatosPt=malloc(NumDatos*sizeof(TipoDato)))
      {
         /* leer datos de los puntos ya que tenemos memoria */
         if(NumDatos!=fread((char
*)*DatosPt,sizeof(TipoDato),(size_t)NumDatos,FEnt))
         {
            free(*DatosPt);
            return 0;
         }
      }
      else return 0; /* falta de memoria */

      return 1; /* terminacion correcta */
   }
   else
      return 0; /* error no se puede abrir fichero */
} /* de Carga */


GuardaTraMat(Trayec,NumPunt,NumTra,NomFich)
PuntoTrayec *Trayec[NUMMAXT];
SinSigno NumPunt[NUMMAXT];
unsigned NumTra;
char *NomFich;
{
   CabezaMat Cabeza;
   FILE *Fich;
   Real w[5];
   char NomVar[10];
   unsigned k;
   SinSigno TotalPt;

   Cabeza.tipo=1100;  /* recorre por filas */
   Cabeza.HayImag=0;


   if(Fich=fopen(NomFich,"w"))
   {

      /* Guardar el numero de trayectorias */
      Cabeza.NumFilas=1;
      Cabeza.NumColum=1;
      Cabeza.TamNom=sprintf(NomVar,"numtrayec")+1;

      fwrite((char *)&Cabeza,sizeof(CabezaMat),1,Fich);
      fwrite(NomVar,sizeof(char),(size_t)Cabeza.TamNom,Fich);
      w[0]=NumTra;
      fwrite((char *)w,sizeof(Real),1,Fich);


      /* guardar todas las trayectorias en una matriz.
          - la primera linea indica el numero de puntos
          - las siguientes los 5 parametros de cada punto */
      for(k=0,TotalPt=0;k<NumTra;k++)
         TotalPt+=NumPunt[k];

      Cabeza.NumFilas=TotalPt+NumTra;
      Cabeza.NumColum=5;
      Cabeza.TamNom=sprintf(NomVar,"trayec")+1;

      fwrite((char *)&Cabeza,sizeof(CabezaMat),1,Fich);
      fwrite(NomVar,sizeof(char),(size_t)Cabeza.TamNom,Fich);
      for(k=0;k<NumTra;k++)
      {
         w[0]=NumPunt[k];  /* indicamos el no. de puntos de la trayec*/
         fwrite((char *)w,sizeof(Real),5,Fich); /* en las 5 columnas*/
         /* escribimos los datos de la trayectoria */
         fwrite((char *)Trayec[k],sizeof(Real),(size_t)(5*NumPunt[k]),Fich);
      }


      fclose(Fich);
      return 1;
   }
   return 0;  /* No se pudo abrir el fichero */

} /* de Guarda Trayectoria en Mat */


GuardaTraSal(p,Trayec,NumPunt,NumTrayec,NomFich)
TParametros p;
PuntoTrayec *Trayec[NUMMAXT];
SinSigno NumPunt[NUMMAXT];
unsigned NumTrayec;
char *NomFich;
{
   FILE *Fich;
   PuntoTrayec *PtTy;
   SinSigno i;

   if(Fich=fopen(NomFich,"w"))
   {
      unsigned k;


      fprintf(Fich,"N1menos=%lu   N1Mas=%lu    N2=%lu\n",
                  p.N1menos,p.N1Mas,p.N2);
      fprintf(Fich,"d1=%7.4G   d2=%7.4G\n",p.d1,p.d2);
      fprintf(Fich,"\nConstantes:\nA=%7.4G  B=%7.4G  C=%7.4G  D=%7.4G\n"
            ,p.A,p.B,p.C,p.D);
      fprintf(Fich,"\nCota para el comando:%lf\n",p.CotaCom);
      fprintf(Fich,"\nPesos en el indice L:\nTiempo=%7.4G\n",p.FacT);
      fprintf(Fich,"Comando=%7.4G \nCoordenada
X1=%7.4G\n",p.FacCom,p.FacX1);
      fprintf(Fich,"Coordenada X2= %7.4G");

      for(k=0;k<NumTrayec;k++)
      {
         fprintf(Fich,"\n\nTRAYECTORIA %u:\n",k+1);
         fprintf(Fich,"(%7s,%7s):%7s,%7s,%7s \n",
```

```
    "X1 ","X2 ","U ","T ","I ");
         for(i=0,   PtTy=Trayec[k];
             i<NumPunt[k];
             i++,   PtTy++)
           fprintf(Fich,"(%7.4G,%7.4G):%7.4G,%7.4G,%7.4G \n",PtTy->x1
  ,PtTy->x2,PtTy->consig,PtTy->tau,PtTy->costo);
      }
      fclose(Fich);
      return 1;
   }
   return 0; /* error */
} /* de Guarda Tra sal*/


#include <math.h>
unsigned EntraIni(p,CdIni,NomF)
TParametros p;
Condiciones CdIni[NUMMAXT];
char *NomF;
{
   FILE *FEnt;
   Real x1ini,x2ini;
   unsigned k;

   if(*NomF)
      /* existe fichero de entrada */
      if(FEnt=fopen(NomF,"r"))
      {
         k=0;
         while(k<NUMMAXT && (fscanf(FEnt,"%lg  %lg",&x1ini,&x2ini)==2))
            {
               if((x1ini<-p.d1*p.N1menos) || (x1ini>p.d1*p.N1Mas)
                  || (fabs(x2ini)>p.d2*p.N2))
                  continue;  /* punto fuera del rango */
               CdIni[k].MaxPun=MAXPUNDEF;
               CdIni[k].SigPlano=x2ini>0 ? 1 : -1;
               CdIni[k].iini=(SinSigno)ROUND(-
CdIni[k].SigPlano*x1ini/p.d1)+p.N1menos;
               CdIni[k].jini=(SinSigno)ROUND(fabs(x2ini)/p.d2);
               k++;
            } /* del for */
         fclose(FEnt);
      }
      else return 0;  /* no se puede abrir fichero */
   else {  /* entrada interactiva de datos */

      /* Presentacion de informacion de rangos */
      printf("\nN1menos=%lu    N1Mas=%lu",p.N1menos,p.N1Mas);
      printf("   d1=%7.4G \nX1min=%7.4G    X1max=%7.4G",
             p.d1,-p.d1*p.N1menos,p.d1*p.N1Mas);
      printf("\nN2=%lu    d2=%7.4G",p.N2,p.d2);
      printf("\nX2min=%7.4G    X2max=%7.4G",-p.d2*p.N2,p.d2*p.N2);

      printf("\nConstantes:\nA=%7.4G  B=%7.4G  C=%7.4G  D=%7.4G\n"
             ,p.A,p.B,p.C,p.D);
      printf("\nCota para el comando:%7.4G",p.CotaCom);
      printf("\nPesos en el indice L:\nTiempo=%7.4G",p.FacT);
      printf("\nComando=%7.4G \nCoordenada X1=%7.4G",p.FacCom,p.FacX1);
      printf("\nCoordenada X2= %7.4G\n");

      printf("\nEntra los puntos iniciales(hasta %u)",NUMMAXT);
      printf("\nEntra 0,0 para terminar");
      k=0;
      while(k<NUMMAXT)
      {
         printf("\n Punto inicial %u: \n X1= ",k+1);
         scanf("%lf",&x1ini);
         printf(" X2= ");
         scanf("%lf",&x2ini);
         if ( x1ini==0 && x2ini==0) break;
         if((x1ini<-p.d1*p.N1menos) || (x1ini>p.d1*p.N1Mas)
            || (fabs(x2ini)>p.d2*p.N2))
         {
            printf("-- Punto fuera de rango --");
            continue;  /* puntp fuera del rango */
         }
         CdIni[k].MaxPun=MAXPUNDEF;
         CdIni[k].SigPlano=x2ini>0 ? 1 : -1;
         CdIni[k].iini=(SinSigno)ROUND(-
CdIni[k].SigPlano*x1ini/p.d1)+p.N1menos;
         CdIni[k].jini=(SinSigno)ROUND(fabs(x2ini)/p.d2);
         k++;
      } /* del while k */
   }  /* del else */
   return k;
}  /* de entra ini */


PreparaFich(numarg,argu,NomF)
int numarg;
char *argu[],NomF[4][13];
{
   char Nombre[9];
   int i,pos;
   NomF[0][0]='\0';  /* inicializa los nombres a vacio */
   NomF[1][0]='\0';
   NomF[2][0]='\0';
   NomF[3][0]='\0';
   Nombre[0]='\0';

   /* Obtener nombre base */
   for(i=1;!*Nombre && i<numarg;i++)
      if (*argu[i]!='-')
      {
         char *aux;
         aux=strchr(argu[i],'.');
         pos=(aux)? (aux-argu[i]):10;
         strncpy(Nombre,argu[i],(pos>8?8:pos));
         Nombre[8]='\0';
      };
   if (*Nombre)
   {  /* existe nombre base para los ficheros */
      strcpy(NomF[0],Nombre);
      strcat(NomF[0],".dat");  /* crea fichero datos */
```

```
        for(i=1;i<numarg;i++)
          /* busca las opciones */
          if (argu[i][0]=='-')
             switch (argu[i][1]) {
             case 'm':
                NomF[1][0]='\0';
                if (strlen(argu[i])>2)
                {
                   strncpy(NomF[1],argu[i]+2,12);
                   NomF[1][12]='\0';
                }
                else {
                   strcpy(NomF[1],Nombre);
                   strcat(NomF[1],".mat");
                   };
                break;
             case 's':
                NomF[2][0]='\0';
                if (strlen(argu[i])>2)
                {
                   strncpy(NomF[2],argu[i]+2,12);
                   NomF[2][12]='\0';
                }
                else {
                   strcpy(NomF[2],Nombre);
                   strcat(NomF[2],".slc");
                   };
                break;
             case 'e':
                NomF[3][0]='\0';
                if (strlen(argu[i])>2)
                {
                   strncpy(NomF[3],argu[i]+2,12);
                   NomF[3][12]='\0';
                }
                else {
                   strcpy(NomF[3],Nombre);
                   strcat(NomF[3],".enc");
                   };
             }; /* del switch el if y el for */
          return 1; /* preparados los nombres */
       } /* de if existe nombre */
       return 0; /* no se encontro nombre base */
    }  /* de PreparaParam */
```

A.1.1.8 iodin.c

```
#include "dinamica.h"
#include <stdio.h>
#include <string.h>

int ObtenerParametros(param,NomF)
TParametros *param;
char *NomF;
{
   FILE *fe;

   if (*NomF)
   { /* Tomamos parametros del fichero */
      if (fe=fopen(NomF,"r"))
      {
         int nument;

         nument=fscanf(fe,"%lu %lu %lu %lg %lg",&param->N1menos,
                &param->N1Mas,&param->N2,&param->d1,&param->d2);
         nument+=fscanf(fe,"%lg %lg %lg %lg",&param->A,&param->B,
      &param->C,&param->D);
         nument+=fscanf(fe,"%lg %lg %lg %lg %lg",&param->CotaCom,
            &param->FacT,&param->FacCom,&param->FacX1,&param->FacX2);
         fclose(fe);
         return (nument==NUMPAR); /* correcto si estan los parametros */
      }
      else
         return 0; /* el fichero no se puede abrir */
   }
   else
   {  /* Entramos datos interectivamente */
      printf("\n\nN1menos= ");
      scanf("%lu",&param->N1menos);

      printf("N1Mas= ");
      scanf("%lu",&param->N1Mas);

      printf("N2 = ");
      scanf("%lu",&param->N2);


      printf("d1= ");
      scanf("%lg",&(param->d1));
      printf("d2= ");
      scanf("%lg",&(param->d2));

      printf("\nConstantes:\n A= ");
      scanf("%lg",&(param->A));
      printf(" B= ");
      scanf("%lg",&(param->B));
      printf(" C= ");
      scanf("%lg",&(param->C));
      printf(" D= ");
      scanf("%lg",&(param->D));

      printf("\nCota para el comando=");
      scanf("%lg",&param->CotaCom);

      printf("\nFactor del tiempo= ");
      scanf("%lg",&param->FacT);
      printf("Factor del comando= ");
      scanf("%lg",&param->FacCom);
      printf("Factor coordenada X1= ");
      scanf("%lg",&param->FacX1);
      printf("Factor coordenada X2= ");
      scanf("%lg",&param->FacX2);
```

```
      printf("\n\n");

      return 1; /* terminacion correcta */
   }  /* del si existe fichero */
} /* de entra parametros */


int Salva(param,puntos,nombre)
TParametros param;
TipoDato *puntos;
char *nombre;
{
   FILE *FSal;

   if(FSal=fopen(nombre,"wb"))
   {
      /* Salvar los parametros */
      fwrite((char *)&param,sizeof(TParametros),1,FSal);

      /* Salvar datos en los puntos */
      fwrite((char *)puntos,sizeof(TipoDato),
         (param.N1menos+param.N1Mas+1)*(param.N2+1),FSal);
      fclose(FSal);
      return 1;
   }
   else
      return 0;  /* No puedo abrir fichero */
}  /* de Salva */


int Muestra(p,puntos,nombre)
TParametros p;
TipoDato *puntos;
char *nombre;

{
#define  X1(i)  ((Real)(i)-(Real)p.N1menos)*p.d1
#define  X2(j)  (-(Real)(j))*p.d2
   SinSigno j,i;
   TipoDato *PtFilaAct,*PtAct;
   FILE *fs;

   if(! (fs=fopen(nombre,"w")))
   {
      printf(" No puedo abrir %s",nombre);
      return 0;
   }

   fprintf(fs,"N1menos=%lu  N1Mas=%lu  N2=%lu\n",
            p.N1menos,p.N1Mas,p.N2);
   fprintf(fs,"d1=%11.4G d2=%11.4G",p.d1,p.d2);
   fprintf(fs,"\nConstantes:\nA=%11.4G  B=%11.4G  C=%11.4G  D=%11.4G\n"
           ,p.A,p.B,p.C,p.D);
   fprintf(fs,"\nCota para el comando:%11.4G\n",p.CotaCom);
   fprintf(fs,"\nPesos en el indice L:\nTiempo=%11.4G\n",p.FacT);
   fprintf(fs,"Comando=%11.4G \nCoordenada X1=%11.4G\n",p.FacCom,p.FacX1);
   fprintf(fs,"Coordenada X2= %11.4G",p.FacX2);


   for(i=0,   PtFilaAct=puntos;
       i<=p.N1Mas+p.N1menos;
       i++,   PtFilaAct+=p.N2+1)
   {  /* recorre las filas */
      fprintf(fs,"\n(%7s,%7s):%11s,%11s,%11s=>%s\n"," X1 "," X2 ",
              " I "," U "," Tau "," jsig ");
      for(j=0,      PtAct=PtFilaAct;
          j<=p.N2;
          j++,     PtAct++)
      {
           fprintf(fs,"(%7.4G,%7.4G):%11.4G,%11.4G,%11.4G =>%lu\n",
              X1(i),X2(j),PtAct->I,PtAct->U,PtAct->Tau,PtAct->jsig);
      }  /* del  for j */
   } /* del for i */
   fclose(fs);
   return 1;
#undef  X1
#undef  X2
} /* de Muestra */


int SalvaMat(param,puntos,nombre)
TParametros param;
TipoDato *puntos;
char *nombre;
{
   FILE *FSal;
   CabezaMat ini ;
   char *NomVar ;
   SinSigno Total,i;
   TipoDato *PtAct;
   Real vdoble;

   ini.tipo=1000;
   ini.HayImag=0;


   if(FSal=fopen(nombre,"w"))
   {
      /* salvar los parametros */

      NomVar="Param";
      ini.NumFilas=1;
      ini.NumColum=14;
      ini.TamNom=strlen(NomVar)+1;

      fwrite((char *)&ini,sizeof(CabezaMat),1,FSal);
      fwrite(NomVar,sizeof(char),(size_t)ini.TamNom,FSal);
      vdoble=param.N1menos;
      fwrite((char *)&vdoble,sizeof(Real),1,FSal);
      vdoble=param.N1Mas;
      fwrite((char *)&vdoble,sizeof(Real),1,FSal);
      vdoble=param.N2;
      fwrite((char *)&vdoble,sizeof(Real),1,FSal);
      fwrite((char *)&param.d1,sizeof(Real),NUMPAR-3,FSal);
```

```
        ini.NumColum=param.N1menos+param.N1Mas+1;
        ini.NumFilas=param.N2+1;
        Total=ini.NumFilas*ini.NumColum;

        /* salvar U */
        NomVar="Umin";
        ini.TamNom=strlen(NomVar)+1;
        fwrite((char *)&ini,sizeof(CabezaMat),1,FSal);
        fwrite(NomVar,sizeof(char),(size_t)ini.TamNom,FSal);
        for(i=1,PtAct=puntos;
              i<=Total;
              i++,PtAct++)
                 fwrite((char *)&PtAct->U,sizeof(Real),1,FSal);

        /* salvar I */
        NomVar="Imin";
        ini.TamNom=strlen(NomVar)+1;
        fwrite((char *)&ini,sizeof(CabezaMat),1,FSal);
        fwrite(NomVar,sizeof(char),(size_t)ini.TamNom,FSal);
        for(i=1,PtAct=puntos;
              i<=Total;
              i++,PtAct++)
                fwrite((char *)&PtAct->I,sizeof(Real),1,FSal);

        /* salvar Tau */
        NomVar="Tmin";
        ini.TamNom=strlen(NomVar)+1;
        fwrite((char *)&ini,sizeof(CabezaMat),1,FSal);
        fwrite(NomVar,sizeof(char),(size_t)ini.TamNom,FSal);
        for(i=1,PtAct=puntos;
              i<=Total;
              i++,PtAct++)
                fwrite((char *)&PtAct->Tau,sizeof(Real),1,FSal);

        /* salvar jsig */
        NomVar="jsig";
        ini.TamNom=strlen(NomVar)+1;
        fwrite((char *)&ini,sizeof(CabezaMat),1,FSal);
        fwrite(NomVar,sizeof(char),(size_t)ini.TamNom,FSal);
        for(i=1,PtAct=puntos;
              i<=Total;
              i++,PtAct++)
        {
           vdoble=PtAct->jsig;
           fwrite((char *)&vdoble,sizeof(Real),1,FSal);
        }


        fclose(FSal);
        return 1;
     }
     else  {
        printf(" ERROR ** No puedo abrir el fichero de escritura");
        return 0;
     }
}  /* de SalvaMat */


PreparaFich(numarg,argu,NomF)
int numarg;
char *argu[],NomF[4][13];
{
   char Nombre[9];
   int i,pos;
   NomF[0][0]='\0';  /* inicializa los nombres a vacio */
   NomF[1][0]='\0';
   NomF[2][0]='\0';
   NomF[3][0]='\0';
   Nombre[0]='\0';

   /* Obtener nombre base */
   for(i=1;!*Nombre && i<numarg;i++)
      if (*argu[i]!='-')
      {
         char *aux;
         pos=(aux=strchr(argu[i],'.'))? aux-argu[i]:10;
         strncpy(Nombre,argu[i],(size_t)(pos>8?8:pos));
         Nombre[8]='\0';
      };
   if (*Nombre)
   {  /* existe nombre base para los ficheros */
      strcpy(NomF[0],Nombre);
      strcat(NomF[0],".dat");  /* crea fichero datos */
      for(i=1;i<numarg;i++)
        /* busca las opciones */
        if (argu[i][0]=='-')
           switch (argu[i][1]) {
           case 'm':
              NomF[1][0]='\0';
              if (strlen(argu[i])>2)
              {
                 strncpy(NomF[1],argu[i]+2,12);
                 NomF[1][12]='\0';
              }
              else {
                 strcpy(NomF[1],Nombre);
                 strcat(NomF[1],".mad");
                 };
              break;
           case 's':
              NomF[2][0]='\0';
              if (strlen(argu[i])>2)
              {
                 strncpy(NomF[2],argu[i]+2,12);
                 NomF[2][12]='\0';
              }
              else {
                 strcpy(NomF[2],Nombre);
                 strcat(NomF[2],".sld");
                 };
              break;
           case 'e':
              NomF[3][0]='\0';
              if (strlen(argu[i])>2)
              {
```

```
                    strncpy(NomF[3],argu[i]+2,12);
                    NomF[3][12]='\0';
                 }
                 else {
                    strcpy(NomF[3],Nombre);
                    strcat(NomF[3],".end");
                    };
              }; /* del switch el if y el for */
      return 1; /* preparados los nombres */
   } /* de if existe nombre */
   return 0; /* no se encontro nombre base */
}  /* de PreparaParam */
```

A.1.1.9 param.c

```
#include "dinamica.h"
#include <stdio.h>
#include <stdlib.h>

main(argc,argv)
int argc;
char *argv[];
{
   FILE *FEnt;
   TParametros p;
   char Nomf[18];

   if (argc==2)
   {
      strcpy(Nomf,argv[1]);
      strcat(Nomf,".dat");
      if(FEnt=fopen(Nomf,"r"))
      {
         /* leer los  parametros */
         if(1!=fread((char *)&p,sizeof(TParametros),1,FEnt))
            return 1;  /* error por falta de datos*/
         fclose(FEnt);


         /* Presentacion de informacion de rangos */
         printf("\nN1menos=%lu    N1Mas=%lu",p.N1menos,p.N1Mas);
         printf("   d1=%7.4G \nX1min=%7.4G    X1max=%7.4G",
                 p.d1,-p.d1*p.N1menos,p.d1*p.N1Mas);
         printf("\nN2=%lu    d2=%7.4G",p.N2,p.d2);
         printf("\nX2min=%7.4G    X2max=%7.4G",-p.d2*p.N2,p.d2*p.N2);

         printf("\nConstantes:\nA=%7.4G  B=%7.4G  C=%7.4G  D=%7.4G\n"
                 ,p.A,p.B,p.C,p.D);
         printf("\nCota para el comando:%7.4G",p.CotaCom);
         printf("\nPesos en el indice L:\nTiempo=%7.4G",p.FacT);
         printf("\nComando=%7.4G \nCoordenada X1=%7.4G",p.FacCom,p.FacX1);
         printf("\nCoordenada X2= %7.4G\n",p.FacX2);

         return 0; /* terminacion correcta */
      }
      else
         printf("\n NO se puede abrir fichero indicado \n");
   }
   else
      printf("\n FALTA nombre del fichero \n");
 return 1; /* error */
}
```

A.1.1.10 procesa1.c

```
#include "dinamica.h"
#include <math.h>

void Procesa(p,puntos)
TParametros p;
TipoDato *puntos;
{
   Real x1,x2,x1pru,LimSup1,LimSup2,CuadraCota;
   TipoDato *PtFilaAct,*PtFilaAnt,*PtAct;

   LimSup1=(p.N1Mas*p.d1)+(p.d1/2);
   LimSup2= (-p.d2*p.N2)-(p.d2/2);
   CuadraCota=p.CotaCom*p.CotaCom;

   for(x1=p.d1*(1.0-(Real)p.N1menos), LimSup1=p.N1Mas*p.d1,
       PtFilaAct=puntos+p.N2+1;
       x1<=LimSup1;
       x1+=p.d1,                    PtFilaAct+=p.N2+1)
   {  /* recorre las filas */
      PtFilaAnt=PtFilaAct-(p.N2+1);
      x1pru=x1-p.d1;
      for(PtAct=PtFilaAct, x2=(x1>p.d1/2)?0.0:(PtAct++,-p.d2);
          x2>=LimSup2;
          x2-=p.d2,       PtAct++)
       /* Recorre la columna */
         { /* Busqueda del minimo para este punto */
            Real  Umin,Upru,Tmin,Tpru,Costomin,Costopru;
            Real x2pru;
            SinSigno jpru,jmin;
            TipoDato *PtPru;

            Costomin=HUGE_VAL;
            jmin=MAX_INT;
            for(jpru=0 ,PtPru=PtFilaAnt
          ,x2pru=(x2<0.0)?0.0:(jpru++,PtPru++,-p.d2);
                 x2pru>=LimSup2;
                 x2pru-=p.d2 ,jpru++ ,PtPru++)
                 {
                   if (PtPru->jsig==MAX_INT)
             continue;
                   Upru=CalculaSumaU(p,x1,x2,x1pru,x2pru);
                   if (!jpru && x1pru<p.d1/2)
            Upru=Upru/2;  /* prueba en punto de muerte */
```

```
        else
           Upru=Upru - PtPru->U;

                   Tpru=CalculaTau(p,x1,x2,x1pru,x2pru);
                   Costopru=IndiceL(p,x1,x2,Upru,Tpru)+ PtPru->I;
                   if (Costopru<Costomin && (Upru*Upru)<CuadraCota)
                   {
                      Umin=Upru;
                      Tmin=Tpru;
                      Costomin=Costopru;
           jmin=jpru;
                   }
                }
              PtAct->U=Umin;
              PtAct->Tau=Tmin;
              PtAct->I=Costomin;
              PtAct->jsig=jmin;
           }  /* del if y del for x2 */
   } /* del for x1 */
} /* de procesa */
```

A.1.2 Segundo Sistema

A.1.2.1 din3d.h

```
/* definicion de tipos */
#include <math.h>
#include "tdatos.h"

/*PD sistema Hilare. Las condiciones son las siguientes:
  - U1>0 para permitir x1 monotona por bandas de x3
  - x1 variable de etapa
  - x1 limitada a valores positivos
  - x3 limitada al rango -pi,pi;
  - Punto final el origen
*/
typedef struct
    {
       Real I,Tau,u1,u2;  /*costo minimo y comandos optimos*/
       char x2sig,x3sig,  /*indice del pto sig. horiz y vert. Por
                     ser char=> maximo de 127 ptos en cada dim.*/
     noCambios;   /*numero de veces que se ha actualizado*/
     } TNodo;

typedef struct
    {
       Real A;          /*Parametro de sagunda ecuacion*/
       char Npx1,Npx2p,Npx2n,Npx3; /*Numero de puntos en cada dimension*/
       LargoSS NumNodos,  /*Numero total nodos en regilla*/
         NoProhi;   /*Numero nodos no prohibidos*/
       Real  dx1,dx2;       /*Discretizacion por no haber maximo*/
       Real  u1MAX,u2MAX;   /*Maximos comandos*/
       /*para la func. de costo, factores del valor absoluto*/
       Real Facx2,Facx3,Facu1,Facu2,FacT;
    } TParametros;

typedef struct
{
   Real x1,x2,x3,Tau,u1,u2,Costo;
} PuntoTrayec ;  /* datos en cada punto de la trayectoria */

#define ROUND(x)  ((fabs((x)-floor(x))>.5)?ceil(x):floor(x))
#define PI (2*atan(1))
/*************** PARAMETROS SISTEMA ***********************/
/*Numero de version*/
#define VERSION  1
#ifdef   MSDOS
#   define VERNUM (VERSION | 0x80 )
#else
#   define VERNUM  (VERSION)
#endif


/*Macros de las formulas*/
#define TAPROX          /*Solo aproximacion en el tiempo*/
#define U1APROX
#define U2APROX
/*para el tiempo hacemos la aproximacion*/
#ifdef TAPROX
#define TAU(x1,x2,x3,x1s,x2s,x3s,NS)  (2*((x2s-x2)-tan(x3)*(x1s-x1))/ \
   (2*ParA+NS->u1*(sin(x3s)-tan(x3)*cos(x3s))))
#else
#endif

#ifdef U1APROX
#define U1(x1,x2,x3,x1s,x2s,x3s,tau,NS)       ((2*(x1s-x1)/(tau)-cos(x3s)* \
   (NS->u1))/cos(x3))
#else
#endif

#ifdef U2APROX
#define U2(x1,x2,x3,x1s,x2s,x3s,tau,NS)       (2*(x3s-x3)/(tau)-(NS->u2))
#else
#endif

#define COSTO(x2,x3,u1,u2,tau,P) ((x2)*P.Facx2+(x3)*P.Facx3+ \
   (u1)*P.Facu1+(u2)*P.Facu2+(tau)*P.FacT)
```

A.1.2.2 crea3d.c

```
#include <stdlib.h>
#include <stdio.h>

#include "din3d.h"

int CreaReg(par,reg)
```

```
TParametros par;
TNodo **reg;
{
   /*funcion que crea la regilla:
     - pide la memoria para reg
     - inicializa los valores
     - pone el punto/s final/es
*/
   LargoSS i;
   TNodo *na;

   /*Calculo del la memoria necesaria*/
      /*Ordenacion  -por x3 (mas rapido)
                    -por x2
                    -por x1 (menos rapido) */

   if(!(*reg=malloc(par.NumNodos*sizeof(TNodo))))
   {
      printf("\nNo se obtuvo memoria para la regilla");
      return 0;
   }


   /*inicializacion de valores*/
   for(i=1, na=*reg; i<=par.NumNodos; i++, na++)
   {
      na->I=HUGE_VAL;
      na->x2sig=na->x3sig=-1;
      na->noCambios=0;
   }

   /*Los puntos finales son los del eje x2=0 en plano x1=0*/
   for(i=0, na=*reg+par.Npx2n*(par.Npx3); i<=par.Npx3; i++, na++)
   {
      na->I=0;
      na->x2sig=na->x3sig=0;
      na->u1=na->u2=na->Tau=0.0;
   }


   return 1;
}
```

A.1.2.3 cal3d.c

```
#include <stdlib.h>

#include "din3d.h"


TLogico Calcula(Par,Reg)
TParametros Par;
TNodo *Reg;
{
   /* nucleo de la programacion dinamica. El proceso de
     divide en tres zonas:
  - Zona A: x3 entre PI/2 y PI : x1 decreciente
  - Zona B: x3 entre -PI/2 y -PI : x1 decreciente
  - Zona C: Rango de x3 entre -PI/2 y PI/2: x1 creciente


*/
   TLogico HayCambios=FALSO;
   Real x1,x2,x3,x1p,x2p,x3p;
   char i2,i1,i3,i2p,i3p;
   LargoSS PtosEta,PtosA,NoProA,PtosB,NoProB,PtosC,NoProC;
   TNodo *pet2,*pet1,*p2p,*pact,*p1p,*ppre;
   Real Imin,F,Taumin,tau,u1min,u1,u2min,u2;
   SinSigno i2min,i3min;

   /*Constantes utiles*/
   Real Pi=PI;
   Real ParA=Par.A;
   Real d1=Par.dx1;
   Real d2=Par.dx2;
   Real d3=2*Pi/(Par.Npx3-1);
   SinSigno tcol3=Par.Npx3;   /*elementos en coluna de x3*/
   SinSigno tfil2=Par.Npx2n+Par.Npx2p+1; /*elementos en una fila de x2*/
   SinSigno tplano=tcol3*tfil2; /*tamanio de un plano*/
   char npx2=Par.Npx2n+Par.Npx2p,npx1=Par.Npx1,npx3=Par.Npx3-1;
   /*Limites x3 de regiones*/
   char limS,limI;
   /*cotas*/
   Real u1max=Par.u1MAX; Real u2max=Par.u2MAX;

   limS=3*(limI=(char)(Par.Npx3/4));
   /*Zona A*/
   for(i1=1, pet1=Reg+(int)tplano*i1, x1=i1*d1,
              p1p=pet1-(int)tplano, x1p=x1-d1, PtosA=NoProA=0;
       i1<=npx1;
       i1++, p1p=pet1, x1p=x1, pet1+=(int)tplano, x1+=d1)
   {
      for(i2=0, pet2=pet1+tcol3*i2, x2=(i2-Par.Npx2n)*d2, PtosEta=0;
          i2<=npx2;
          i2++, pet2+=tcol3, x2+=d2)
         for(i3=0, pact=pet2+i3, x3=-Pi+i3*d3;
             i3<=limI;
             i3++, pact++, x3+=d3, PtosA++)
      /*Tenemos situado el punto actual*/
      {
            for(i2p=0, p2p=p1p+tcol3*i2p, x2p=(i2p-Par.Npx2n)*d2,
                  Imin=pact->I;
                i2p<=npx2;
                i2p++, p2p+=tcol3, x2p+=d2)
               for(i3p=0, ppre=p2p+i3p, x3p=-Pi+i3p*d3;
                     i3p<=limI;
                     i3p++, ppre++, x3p+=d3)
         /*Ya tenemos los dos puntos.*/
         if(ppre->x2sig>-1)  /*Solo si el sig.  es permitido*/
         {
            /*Calculo de los comandos para hacer la transicion*/
            /*tiempo no negativo*/
            if((tau=TAU(x1,x2,x3,x1p,x2p,x3p,ppre))<0)
```

```
                /*break; ya que tau no depende de tip continue; */
          continue;
               /*U1 avance*/
               if((u1=U1(x1,x2,x3,x1p,x2p,x3p,tau,ppre))>u1max || u1<0)
                  break; /*ya que ql no depende de tip continue; */
               /*U2 rotacion*/
               if((u2=U2(x1,x2,x3,x1p,x2p,x3p,tau,ppre))>u2max || u2<-u2max)
                  continue;
               /*Se trata de paso valido, Pasamos a calcular su costo*/
               F=COSTO(x2,x3,u1,u2,tau,Par)+ppre->I;
               if(F<Imin)
               {
                  Imin=F; i2min=i2p; i3min=i3p;
                  Taumin=tau; u1min=u1; u2min=u2;
               }
            }
            if(Imin<pact->I)  /*para funcionar en varias pasadas*/
            {
               pact->I=Imin; pact->x2sig=i2min; pact->x3sig=i3min;
               pact->Tau=Taumin; pact->u1=u1min; pact->u2=u2min;
         pact->noCambios++; HayCambios=VERDAD;
               NoProA++; PtosEta++;
            }
         }
         printf("\nEtapa i1=%d\t\tNo. de puntos validos=%lu",i1,PtosEta);
      }
   /*Fin de zona A*/
      printf("\nTerminada zona A.");
      printf("\nNo. puntos no prohibidos=%lu de %lu",NoProA,PtosA);

      /*Zona B*/
      for(i1=1, pet1=Reg+(int)tplano*i1, x1=i1*d1,
                p1p=pet1-(int)tplano, x1p=x1-d1, PtosB=NoProB=0;
          i1<=npx1;
          i1++, p1p=pet1, x1p=x1, pet1+=(int)tplano, x1+=d1)
      {
         for(i2=0, pet2=pet1+tcol3*i2, x2=(i2-Par.Npx2n)*d2, PtosEta=0;
             i2<=npx2;
             i2++, pet2+=tcol3, x2+=d2)
            for(i3=limS, pact=pet2+i3, x3=-Pi+i3*d3;
                i3<=npx3;
                i3++, pact++, x3+=d3, PtosB++)
         /*Tenemos situado el punto actual*/
         {
               for(i2p=0, p2p=p1p+tcol3*i2p, x2p=(i2p-Par.Npx2n)*d2,
                     Imin=pact->I;
                   i2p<=npx2;
                   i2p++, p2p+=tcol3, x2p+=d2)
                  for(i3p=limS, ppre=p2p+i3p, x3p=-Pi+i3p*d3;
                      i3p<=npx3;
                      i3p++, ppre++, x3p+=d3)
            /*Ya tenemos los dos puntos.*/
            if(ppre->x2sig>-1)  /*Solo si el sig.  es permitido*/
            {
               /*Calculo de los comandos para hacer la transicion*/
               /*tiempo no negativo*/
               if((tau=TAU(x1,x2,x3,x1p,x2p,x3p,ppre))<0)
                 /*break; ya que tau no depende de tip continue; */
          continue;
               /*U1 avance*/
               if((u1=U1(x1,x2,x3,x1p,x2p,x3p,tau,ppre))>u1max || u1<0)
                  break; /*ya que ql no depende de tip continue; */
               /*U2 rotacion*/
               if((u2=U2(x1,x2,x3,x1p,x2p,x3p,tau,ppre))>u2max || u2<-u2max)
                  continue;
               /*Se trata de paso valido, Pasamos a calcular su costo*/
               F=COSTO(x2,x3,u1,u2,tau,Par)+ppre->I;
               if(F<Imin)
               {
                  Imin=F; i2min=i2p; i3min=i3p;
                  Taumin=tau; u1min=u1; u2min=u2;
               }
            }
            if(Imin<pact->I)  /*para funcionar en varias pasadas*/
            {
               pact->I=Imin; pact->x2sig=i2min; pact->x3sig=i3min;
               pact->Tau=Taumin; pact->u1=u1min; pact->u2=u2min;
         pact->noCambios++; HayCambios=VERDAD;
               NoProB++; PtosEta++;
            }
         }
         printf("\nEtapa i1=%d\t\tNo. de puntos validos=%lu",i1,PtosEta);
      }
   /*Fin de zona B*/
      printf("\nTerminada zona B.");
      printf("\nNo. puntos no prohibidos=%lu de %lu",NoProB,PtosB);

      /*Zona C*/
      for(i1=npx1-1, pet1=Reg+(int)tplano*i1, x1=i1*d1,
                p1p=pet1+(int)tplano, x1p=x1+d1, PtosC=NoProC=0;
          i1>=0;
          i1--, p1p=pet1, x1p=x1, pet1-=(int)tplano, x1-=d1)
      {
         for(i2=0, pet2=pet1+tcol3*i2, x2=(i2-Par.Npx2n)*d2, PtosEta=0;
             i2<=npx2;
             i2++, pet2+=tcol3, x2+=d2)
            for(i3=limI+1, pact=pet2+i3, x3=-Pi+i3*d3;
                i3<limS;
                i3++, pact++, x3+=d3, PtosC++)
         /*Tenemos situado el punto actual*/
         {
               for(i2p=0, p2p=p1p+tcol3*i2p, x2p=(i2p-Par.Npx2n)*d2,
                     Imin=pact->I;
                   i2p<=npx2;
                   i2p++, p2p+=tcol3, x2p+=d2)
                  for(i3p=limI, ppre=p2p+i3p, x3p=-Pi+i3p*d3;
                      i3p<=limS;
                      i3p++, ppre++, x3p+=d3)
            /*Ya tenemos los dos puntos.*/
            if(ppre->x2sig>-1)  /*Solo si el sig.  es permitido*/
            {
               /*Calculo de los comandos para hacer la transicion*/
               /*tiempo no negativo*/
               if((tau=TAU(x1,x2,x3,x1p,x2p,x3p,ppre))<0)
                 /*break; ya que tau no depende de tip continue; */
          continue;
```

```
                /*U1 avance*/
                if((u1=U1(x1,x2,x3,x1p,x2p,x3p,tau,ppre))>u1max || u1<0)
                   break; /*ya que q1 no depende de tip continue; */
                /*U2 rotacion*/
                if((u2=U2(x1,x2,x3,x1p,x2p,x3p,tau,ppre))>u2max || u2<-u2max)
                   continue;
                /*Se trata de paso valido, Pasamos a calcular su costo*/
                F=COSTO(x2,x3,u1,u2,tau,Par)+ppre->I;
                if(F<Imin)
                {
                   Imin=F; i2min=i2p; i3min=i3p;
                   Taumin=tau; u1min=u1; u2min=u2;
                }
             }
             if(Imin<pact->I)  /*para funcionar en varias pasadas*/
             {
                pact->I=Imin; pact->x2sig=i2min; pact->x3sig=i3min;
                pact->Tau=Taumin; pact->u1=u1min; pact->u2=u2min;
          pact->noCambios++; HayCambios=VERDAD;
                NoProC++; PtosEta++;
             }
          }
          printf("\nEtapa i1=%d\t\tNo. de puntos validos=%lu",i1,PtosEta);
       }
    /*Fin de zona C*/
       printf("\nTerminada zona C.");
       printf("\nNo. puntos no prohibidos=%lu de %lu",NoProC,PtosC);

       Par.NoProhi=NoProC+NoProB+NoProA;
       printf("\nNo. puntos no prohibidos totales=%lu de %lu(%lu)",
          Par.NoProhi,PtosA+PtosB+PtosC,Par.NumNodos);
       return HayCambios;
    }
```

A.1.2.4 io3d.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "din3d.h"

int LeePar(par,NomFP,Todo)
TParametros *par;
char *NomFP;
TLogico Todo;
{
   /*Rutina que lee los parametros, de fichero o interactivamente*/
   FILE *fp;
   char c;
   int  tmp;

   if(*NomFP)
   {
      /*existe nombre de fichero -> lo intentamos abrir */
      if(!(fp=fopen(NomFP,"r")))
      {
  printf("\nNO se pudo abrir fichero de parametros");
  return 0;
      }
      fscanf(fp,"%lg",&par->A);
      fscanf(fp,"%d",&tmp); par->Npx1=tmp;
      fscanf(fp,"%d",&tmp); par->Npx2n=tmp;
      fscanf(fp,"%d",&tmp); par->Npx2p=tmp;
      fscanf(fp,"%d",&tmp); par->Npx3=tmp;
      fscanf(fp,"%lg",&par->dx1);
      fscanf(fp,"%lg",&par->dx2);
      fscanf(fp,"%lg",&par->u1MAX);
      fscanf(fp,"%lg",&par->u2MAX);
      fscanf(fp,"%lg",&par->FacT);
      fscanf(fp,"%lg",&par->Facx2);
      fscanf(fp,"%lg",&par->Facx3);
      fscanf(fp,"%lg",&par->Facu1);
      fscanf(fp,"%lg",&par->Facu2);
      par->NumNodos=(par->Npx1+1)*(par->Npx2n+par->Npx2p+1)*
  (par->Npx3);
      return 1;
   }


   /*menu para la mdificacion de los parametros*/
   do {
      if(!Todo)
      {
  printf("\n\n\nA-Parametro A=\t%lg",par->A);
  printf("\n\n1-Numero de puntos dimension x1=\t%d",par->Npx1);
  printf("\n2-Numero de puntos dimension x2 negativa=\t%d"
     ,par->Npx2n);
  printf("\n3-Numero de puntos dimension x2 positiva=\t%d"
     ,par->Npx2p);
  printf("\n4-Numero de puntos dimension x3=\t%d",par->Npx3);
  printf("\n6-Discretizacion en x1=\t%lg",par->dx1);
  printf("\n7-Discretizacion en x2=\t%lg",par->dx2);
  printf("\n\n8-Maximo comando u1=\t%lg",par->u1MAX);
  printf("\n9-Maximo comando u2=\t%lg",par->u2MAX);
  printf("\n\nt-Factor tiempo en costo=\t%lg",par->FacT);
  printf("\nu-Factor comando u1=\t%lg",par->Facu1);
  printf("\nd-Factor comando u2=\t%lg",par->Facu2);
  printf("\nx-Factor x2 en costo=\t%lg",par->Facx2);
  printf("\ny-Factor x3 en costo=\t%lg",par->Facx3);
  printf("\n\nPulsa inicial del parametro a modificar.(0 salir)");
  while((c=getchar())<' ');
      }
      else c='a';
      switch (c|0x20)
      {
      case 'a':
  printf("\nParametro A=\t");
  scanf("%lg",&par->A); if(!Todo) break;
      case '1':
  do {
         printf("\nNumero de puntos dimension x1=\t");
     scanf("%d",&tmp);
  } while(tmp<=0 || tmp>=128); par->Npx1=tmp; if(!Todo) break;
```

```
        case '2':
    do {
       printf("\nNumero de puntos negativos dimension x2=\t");
       scanf("%d",&tmp);
    } while(tmp<=0 || tmp>=128); par->Npx2n=tmp;
    if(!Todo) break;
        case '3':
    do {
       printf("\nNumero de puntos positivos dimension x2=\t");
       scanf("%d",&tmp);
    } while(tmp<=0 || tmp>=128); par->Npx2p=tmp;
    if(!Todo) break;
        case '4':
    do {
       printf("\nNumero de puntos dimension x3=\t");
       scanf("%d",&tmp);
    } while(tmp<=1 || tmp>=128);
    par->Npx3=(char)(tmp/2)*2+1; /*asegurar numero impar*/
    if(!Todo) break;
        case '6':
           printf("\nDiscretizacion en x1=\t");
    scanf("%lg",&par->dx1);
    if(!Todo) break;
        case '7':
           printf("\nDiscretizacion en x2=\t");
    scanf("%lg",&par->dx2);
    if(!Todo) break;
        case '8':
           printf("\nMaximo u1 =\t");
    scanf("%lg",&par->u1MAX); if(!Todo) break;
        case '9':
           printf("\nMaximo u2=\t");
    scanf("%lg",&par->u2MAX); if(!Todo) break;
        case 't':
           printf("\nFactor tiempo en costo=\t");
    scanf("%lg",&par->FacT); if(!Todo) break;
        case 'u':
           printf("\nFactor u1=\t");
    scanf("%lg",&par->Facu1); if(!Todo) break;
        case 'd':
           printf("\nFactor u2=\t");
    scanf("%lg",&par->Facu2); if(!Todo) break;
        case 'x':
           printf("\nFactor x2 en costo=\t");
    scanf("%lg",&par->Facx2); if(!Todo) break;
        case 'y':
           printf("\nFactor x3 en costo=\t");
    scanf("%lg",&par->Facx3); if(!Todo) break;
        } /*fin del switch*/
        Todo=FALSO;
    } while(c!='0');
    par->NumNodos=(par->Npx1+1)*(par->Npx2n+par->Npx2p+1)*(par->Npx3+1);
    return 1;
}

int SalvaReg(Par,Reg,NFil)
TParametros Par;
TNodo *Reg;
char *NFil;
{
    /*Rutina para salvar la regilla*/
    FILE *fr;
    char NFtmp[30],VerNum=VERNUM;

    if(*NFil)     /*Existe nombre*/
    {
        if(!(fr=fopen(NFil,"wb")))
        {
    /*No se pudo abrir el fichero por defecto vamos a tmp*/
    strcpy(NFtmp,"/tmp/");
    strcat(NFtmp,NFil);
    if(!(fr=fopen(NFtmp,"wb")))
    {
       printf("\nNo se pudo abrir ni el fichero de emergencia");
       return 0;
    }
        }
    }
    else       /*Preguntamos el nombre*/
    {
        do {
    *NFtmp=0;
    printf("\nNombre del fichero:");
    scanf("%s",NFtmp);
        } while (*NFtmp && !(fr=fopen(NFtmp,"wb")));
        if(!*NFtmp)
        {
    printf("\nNo se especifica nombre: no se salva datos");
    return 0;
        }
        if(!(fr=fopen(NFtmp,"wb")))
        {
           printf("\nNo se pudo abrir ni el fichero de emergencia");
           return 0;
        }
    }

    /*Comenzamos a salvar*/
    /*primero el numero de la version*/
    fwrite(&VerNum,sizeof(char),1,fr);

    fwrite(&Par,sizeof(TParametros),1,fr);
    fwrite(Reg,sizeof(TNodo),Par.NumNodos,fr);

    fclose(fr);
    return 1;
}
```

A.1.2.5 prin3d.c

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include "din3d.h"

TLogico Calcula();

main(argc,argv)
int argc;
char *argv[];
{
   TParametros Par;
   TNodo *Regilla;
   int pasada,MaxPas=1;
   TLogico HayCambio;

   char NFPar[20],NFReg[20];
   *NFPar=*NFReg='\0';

   if(argc>=2)  strcpy(NFPar,argv[2]);
   LeePar(&Par,NFPar,VERDAD);
   if(!CreaReg(Par,&Regilla))
      return 1;
   if((argc>=1) && sscanf(argv[1],"%d",&pasada))
      MaxPas=pasada;
   printf("\nMaxima pasada permitida es %d",pasada);
   for(pasada=1, HayCambio=VERDAD; HayCambio && pasada<=MaxPas; pasada++)
   {
      printf("\n\n Pasada %d", pasada);
      HayCambio=Calcula(Par,Regilla);
   }
   printf("\n Numero total de pasadas %d",pasada);
   if(argc>=3)  strcpy(NFReg,argv[3]);
   SalvaReg(Par,Regilla,NFReg); /*para que pida todos los par*/

   return 0;
}
```

A.1.2.6 sacmat3d.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "din3d.h"

/*Programa para sacar informacion a matlab de fichero dat*/


void Procesa(fd,fm)
FILE *fd,*fm;
{
   char version;
   TParametros Par;
   CabezaMat Cab;
   char *NomInd="Indices",*NomPar="Param";
   Real TemReal;
   fpos_t posDat;
   LargoSS i;
   TNodo na;

   /*funcion que leera el fichero de datos he ira creando el mat*/
   /*Primero leemos el numero de version y parametros*/
   fread(&version,sizeof(char),1,fd);
   fread(&Par,sizeof(TParametros),1,fd);

   /*creamos cabecera de  matriz*/
   Cab.tipo=1000;
   Cab.HayImag=0;

   /*Salvamos matriz con parametros*/
   Cab.NumFilas=1;
   Cab.NumColum=16;  /*numero total de parametros*/
   Cab.TamNom=strlen(NomPar)+1;
   fwrite(&Cab,sizeof(CabezaMat),1,fm);
   fwrite(NomPar,sizeof(char),Cab.TamNom,fm);
   fwrite(&Par.A,sizeof(Real),1,fm);
   TemReal=Par.Npx1; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.Npx2n; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.Npx2p; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.Npx3; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.NumNodos; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.NoProhi; fwrite(&TemReal,sizeof(Real),1,fm);
   fwrite(&Par.dx1,sizeof(Real),9,fm);
   /*ya esta la matriz de parametros*/

   fgetpos(fd,&posDat);  /*tomamos nota del principio de los datos*/

   Cab.NumFilas=Par.NumNodos;
   Cab.NumColum=1;
   Cab.TamNom=strlen(NomInd)+1;

   fwrite(&Cab,sizeof(CabezaMat),1,fm);
   fwrite(NomInd,sizeof(char),Cab.TamNom,fm);
   for(i=1; i<=Par.NumNodos; i++)
   {
      fread(&na,sizeof(TNodo),1,fd);
      TemReal=na.x2sig+na.x3sig*1000+na.noCambios*1000000;
      fwrite(&TemReal,sizeof(Real),1,fm);
   }

#ifdef PRUEBA
   /*Volvemos al punto inicial*/
   fsetpos(fd,&posDat);

   Cab.TamNom=strlen(NomTau)+1;
   fwrite(&Cab,sizeof(CabezaMat),1,fm);
   fwrite(NomInd,sizeof(char),Cab.TamNom,fm);

   for(i=1; i<=Par.NumNodos; i++)
   {
      fread(&na,sizeof(TNodo),1,fd);
      IndMasCam=na.h1sig+na.tisig*1000+na.noCambios*1000000;
      fwrite(IndMasCam,sizeof(Real),1,fm);
   }
```

```
#endif


}
main(argc,argv)
int argc;
char *argv[];
{
   FILE *fd,*fm;

   if((argc<1) || !(fd=fopen(argv[1],"rb")))
   {
      printf("\n Necesario especificar fichero de datos");
      exit(1);
   }
   if((argc<2) || !(fm=fopen(argv[2],"wb")))
   {
      printf("\n No se puede abrir el fichero matlab");
      exit(2);
   }
   Procesa(fd,fm);

   fclose(fd);
   fclose(fm);
   return 0;
}
```

A.1.2.7 tray3d.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "din3d.h"
#define  VE 3

/*Programa para sacar informacion a matlab de fichero dat*/
#define MAXPTINI 28

int  LeeDatos(fd,par,reg)
FILE *fd;
TParametros *par;
TNodo **reg;
{
   char version;

   /*funcion que leera el fichero de datos */
   /*Primero leemos el numero de version y parametros*/
   fread(&version,sizeof(char),1,fd);
   fread(par,sizeof(TParametros),1,fd);

   /*Pedimos memoria*/
   if(!(*reg=malloc(par->NumNodos*sizeof(TNodo))))
   {
      printf("\nNo se pudo obtener memoria para regilla");
      return 0;
   }
   return (fread(*reg,sizeof(TNodo),par->NumNodos,fd))==par->NumNodos;

}

char LeePtIni(NomIni,par,ptini)
char *NomIni;
TParametros par;
Real *ptini;
{
   /*funcion que lee los puntos iniciales*/
   Real *pt;
   char k;
   FILE *fi;

   if(*NomIni)  /*se ha especificado nombre*/
   {
      if(!(fi=fopen(NomIni,"r")))
      {
     fprintf(stderr,"\nNo se puede abrir fichero de iniciales");
     return 0;
      }
      for(pt=ptini, k=0; k<=3*MAXPTINI &&
   (fscanf(fi,"%lg",pt)!=EOF); k++, pt++);
      return k/3;
   }
   printf("\nEntra los puntos iniciales(hasta %u)",MAXPTINI);
   printf("\nEntra un carcter para terminar");
   for(k=0, pt=ptini; k<MAXPTINI; k++)
   {
      printf("\n Punto inicial %u:\n",k+1);
      printf("x1=");
      if(!scanf("%lf",pt)) break;
      pt++;
      printf("x2=");
      if(!scanf("%lf",pt)) break;
      pt++;
      printf("x3=");
      if(!scanf("%lf",pt)) break;
      pt++;
   } /* del while k */
   return k;
}


SinSigno PtosPro(Ptos)
Real *Ptos;
{
   Real *pt,xi,xf,Xmin[VE],Xmax[VE];
   SinSigno nptos,k,i,resto,pot;
   TLogico Correcto;

   /*Deberia explorar todas las esquinas de las regiones */
   do
   {
      Correcto= VERDAD;
```

```
        printf("\n Entra nuevas coordenadas:");
        for(i=1; i<=VE; i++)
        {
    printf("\nPara X(%hu):\n Valor minimo=",i);
    scanf("%lg",&xi);
    printf(" Valor maximo=");
    scanf("%lg",&xf);
    if  (xi>xf)
    {
        printf("\n ERROR: Valor minimo ha de ser <= que maximo");
       Correcto=FALSO;
       break;
    }
    Xmin[i-1]=xi;
    Xmax[i-1]=xf;
        }
    } while(! Correcto);

    /*en el caso general tentremos 2 elevado a VE ptos iniciales
      y el central*/

    nptos=0;

    /*puntos extremos del hipercubo optimizando extremos iguales*/
    for(k=0, pt=Ptos; k<27; k++)
    {
       nptos++;
       for(i=0; i<VE; i++, pt++)
       {
   pot=pow(3.0,(Real)i);
   resto=(k/pot)%(3);
   if(Xmax[i]==Xmin[i])
   {
      *pt=Xmax[i];
      if (resto==0) k+=pot*2; /*saltamos  maximo directamente*/
   }
   else if(resto==0)
        *pt=Xmin[i];
  else if(resto==1)
     *pt=Xmax[i];
   else
     *pt=(Xmin[i]+Xmax[i])/2;

       }
    }

    return nptos;
}

void CalSal(par,reg,ptini,Nptos,fm)
TParametros par;
TNodo *reg;
Real *ptini;
char  Nptos;
FILE *fm;
{
#define DIRNODO(ix1,ix2,ix3)  (reg+((ix1)*(par.Npx2n+par.Npx2p+1)+ \
    (ix2))*(par.Npx3)+(ix3))
    /*Funcion que calcula y salva trayectorias*/
    CabezaMat Cab;
    char *NomTra="trayec";
    Real *pt,w[6],Costo;  /*vector de la anchura de matriz*/
    char ix1,ix2,ix3,ii1,ii2,ii3,p,i,nptos;
    SinSigno Numfil;
    TNodo *na;
    Real Pi=PI;
    char limI,limS;

    limS=3*(limI=(char)(par.Npx3/4));


 /*creamos cabecera de  matriz*/
    Cab.tipo=1100;
    Cab.HayImag=0;

    /*Salvamos matriz con parametros*/
    Cab.NumFilas=1;
    Cab.NumColum=2*3+1;  /*numero total de parametros*/
    Cab.TamNom=strlen(NomTra)+1;
    /*Salvamos Cabeza incompleta*/
    fwrite(&Cab,sizeof(CabezaMat),1,fm);
    fwrite(NomTra,sizeof(char),Cab.TamNom,fm);

    for(i=1, pt=ptini, Numfil=0; i<=Nptos; i++, pt+=3)
    {
       /*Comprobamos pto inicial dentro de limites*/
    /*   if( (pt[0]<0)  || (pt[0]>par.dx1*par.Npx1) ||
    (pt[1]<-par.dx2*par.Npx2n) || (pt[1]>par.dx2*par.Npx2p) ||
    (pt[2]<-Pi) || (pt[2]>Pi) )
   continue;  pasamos de ese punto*/
       if( (pt[0]<0) || (pt[0]>par.dx1*par.Npx1) ) continue;
  if ((pt[1]<-par.dx2*par.Npx2n) || (pt[1]>par.dx2*par.Npx2p))
     continue;
  if( (pt[2]<-Pi) || (pt[2]>Pi) )
   continue; /* pasamos de ese punto*/
       ii1=ix1=ROUND(pt[0]/par.dx1);
       ii2=ix2=ROUND(pt[1]/par.dx2)+par.Npx2n;
       ii3=ix3=ROUND(pt[2]/2/Pi*(par.Npx3-1))+(int)(par.Npx3/2);

       /*Contamos numero de untos de la trayectoria*/
       for(nptos=1, na=DIRNODO(ix1,ix2,ix3);
    na->x3sig!=-1 && na->x3sig!=0 && nptos<1000; /*evita bucles*/
    nptos++, na=DIRNODO(ix1,ix2,ix3))
       {
    if(na->x2sig>0)   /*en caso contrario estamos en el mismo plano*/
    {
       if(ix3>limI && ix3<limS)
          ix1++;
       else
          ix1--;
       ix2=na->x2sig;
    }
    else
       ix2=-na->x2sig;
```

```
   ix3=na->x3sig;
       }

       /*Ya podemos salvar la tryectoria completa*/
       ix1=ii1; ix2=ii2; ix3=ii3;
       na=DIRNODO(ix1,ix2,ix3);

       /*debemos salvar 3 filas iniciales de la trayectoria*/
       w[0]=nptos; w[1]=3; w[2]=3; w[3]=na->I;
       fwrite((void *)w,sizeof(Real),7,fm); Numfil++;
       w[0]=par.A;
       fwrite((void *)w,sizeof(Real),7,fm); Numfil++;
       w[0]=par.FacT; w[1]=par.Facu1; w[2]=par.Facu2;
       w[3]=par.Facx2; w[4]=par.Facx3;
       fwrite((void *)w,sizeof(Real),7,fm); Numfil++;

       /*Comenzamos a salvar los puntos*/
       for(p=1, Costo=0 ;p<=nptos; p++)
       {
   w[0]=ix1*par.dx1;  /*x1 pto actual*/
   w[1]=(ix2-par.Npx2n)*par.dx2;  /*x2 pto actual*/
   w[2]=ix3*2*Pi/(par.Npx3-1)-Pi;
   w[3]=na->u1; w[4]=na->u2; w[5]=na->Tau;
   w[6]=Costo;
        fwrite((void *)w,sizeof(Real),7,fm); Numfil++;
   /*Pasamos al nodo siguiente*/
   Costo+=COSTO(w[1],w[2],w[3],w[4],w[5],par);
   if(na->x2sig>0)   /*en caso contrario estamos en el mismo plano*/
   {
      if(ix3>limI && ix3<limS)
         ix1++;
      else
         ix1--;
      ix2=na->x2sig;
   }
   else
      ix2=-na->x2sig;
   ix3=na->x3sig;
   na=DIRNODO(ix1,ix2,ix3);
       }
    } /*bucle de ptos iniciales*/

    rewind(fm); /*volvemos al principio para corregir la cabeza*/
    if(fseek (fm, 0L, SEEK_SET))
    {
       fprintf(stderr,"\nProblema al volver al principio\n");
       return;
    }

    Cab.NumFilas=Numfil;
    printf("\nNumero de filas %d\n",Cab.NumFilas);
    /*Salvamos Cabeza completa*/
    if(!fwrite(&Cab,sizeof(CabezaMat),1,fm))
    {
       fprintf(stderr,"\nProblema al machacar cabeza\n");
    }
}

main(argc,argv)
int argc;
char *argv[];
{
   FILE *fd,*fm;
   TNodo *reg;
   TParametros par;
   char NomIni[20];
   SinSigno Nptos;
   Real ptini[MAXPTINI*3];

   if((argc<1) || !(fd=fopen(argv[1],"rb")))
   {
      fprintf(stderr,"\n Necesario especificar fichero de datos\n");
      exit(1);
   }
   if((argc<2) || !(fm=fopen(argv[2],"wb")))
   {
      fprintf(stderr,"\n No se puede abrir el fichero matlab\n");
      exit(2);
   }
   *NomIni=0;
   if(argc>=3)
      strcpy(NomIni,argv[3]);
   if( !LeeDatos(fd,&par,&reg))
      exit(1);

   printf("\nProbar puntos de la region ([s]/n)?");
   if((getchar()|0x20)!='n')
      Nptos=PtosPro(ptini);
   else
      Nptos=LeePtIni(NomIni,par,ptini);
   CalSal(par,reg,ptini,Nptos,fm);


   fclose(fd);
   fclose(fm);
   return 0;
}
```

A.1.3 Tercer Sistema

A.1.3.1 din3d.h

```
/* definicion de tipos */
#include <math.h>
#include "tdatos.h"


typedef struct
   {
```

```
      Real I,Tau,ql,qc;  /*costo minimo y comandos optimos*/
      char hisig,tisig,  /*indice del pto sig. horiz y vert. Por
                   ser char=> maximo de 127 ptos en cada dim.*/
    noCambios;   /*numero de veces que se ha actualizado*/
    } TNodo;

typedef struct
   {
      Real h0;         /*Diff. altura entre tanques*/
      Real h2des,Tides;      /*punto final deseado*/
      char Nph1,Nph2,NpTi;   /*Numero de puntos en cada dimension*/
      LargoSS NumNodos,  /*Numero total nodos en regilla*/
        NoProhi;   /*Numero nodos no prohibidos*/
      Real  dTi;       /*Discretizacion en Tita, por no haber maximo*/
      Real  qlMAX,qcMAX;   /*Maximos comandos*/
      /*para la func. de costo, factores del valor absoluto*/
      Real FacT,FacQl,FacQc,FacH1,FacTi;
   } TParametros;

typedef struct
{
  Real h1,h2,Ti,Tau,ql,qc,Costo;
} PuntoTrayec ;  /* datos en cada punto de la trayectoria */

#define ROUND(x)  ((fabs((x)-floor(x))>.5)?ceil(x):floor(x))
/*************** PARAMETROS SISTEMA ***********************/
/*Numero de version*/
#define VERSION  1
#ifdef __MSDOS
#   define VERNUM (VERSION | 0x80 )
#else
#   define VERNUM  (VERSION)
#endif


#define H1MAX  10.0
#define H2MAX  6.0
/*#define R1    1.0/1.48  constantes orificios usadas en NN*/
#define R1  2.8   /*doble que antes*/
#define R2       R1
#define CP       1.0     /*constantes de calor*/
#define CS       1.5
#define RT       1.0

/*Macros de las formulas*/
#define TAPROX          /*Solo aproximacion en el tiempo*/
#define QLAPROX
#define QCAPROX
/*para el tiempo hacemos la aproximacion*/
#ifdef TAPROX
#define TAU(h1,h2,ti,h1s,h2s,tis,NS)  (2*(h2s-h2)/ \
   (R1*(h1s-h2s+h1-h2+2*h0)-R2*(h2+h2s)))
#else
#define TAU(h1,h2,ti,h1s,h2s,tis,NS)  ((2*(h2s-h2)+NS->Tau* \
   (-R1*(h1s-h2s+h0)+R2*h2s))/(R1*(h1-h2+h0)-R2*h2))
#endif

#ifdef QLAPROX
#define QL(h1,h2,ti,h1s,h2s,tis,tau,NS)       (2*(h1s-h1)/(tau)+R1*((h1-
h2+h0)+ \
   (h1s-h2s+h0))-(NS->ql))
#else
#define QL(h1,h2,ti,h1s,h2s,tis,tau,NS)       ((2*(h1s-h1)+R1*((tau)*(h1-
h2+h0)+ \
   NS->Tau*(h1s-h2s+h0))-(NS->Tau)*(NS->ql))/(tau))
#endif

#ifdef QCAPROX
#define QC(h1,h2,ti,h1s,h2s,tis,tau,NS)       (2*(tis-ti)/(tau)+((ti+tis) \
   /RT+CP*R1*(ti*(h1-h2+h0)+tis*(h1s-h2s+h0))- \
   CP*R2*(ti*h2+tis*h2s))/CS-(NS->qc))
#else
#define QC(h1,h2,ti,h1s,h2s,tis,tau,NS)       ((2*(tis-ti)+((((tau)*ti+NS-
>Tau*tis) \
   /RT)+CP*R1*((tau)*ti*(h1-h2+h0)+(NS->Tau)*tis*(h1s-h2s+h0))- \
   CP*R2*((tau)*ti*h2+(NS->Tau)*tis*h2s))/CS-(NS->Tau)*(NS->qc))/(tau))
#endif

#define COSTO(h1,ti,qc,ql,tau,P) ((h1)*P.FacH1+(ti)*P.FacTi+ \
   (qc)*P.FacQc+(ql)*P.FacQl+(tau)*P.FacT)
```

A.1.3.2 cal3d.c

```
#include <stdlib.h>

#include "din3d.h"


TLogico Calcula(Par,Reg,Var,EtaDes)
TParametros Par;
TNodo *Reg;
char *Var,EtaDes;
{
   /* nucleo de la programacion dinamica. El proceso de
     divide en tres zonas:
        - Zona A: en h2 decreciente ,desde la etapa siguiente del
                pto deseado hasta la etapa H2MAX. Barremos h1 desde
                la variedad hasta 0.
        - Zona B: zona de h2 creciente desde H2MAX hasta 0. h1
                barre desde H1MAX hasta la variedad.
        - Zona C: otra vez en h2 creciente para terminar la zona
                no calculada: desde 0 hasta la etapa del pto deseado.
                h1 barriendo desde 0 hasta la variedad */
/*

             *--------------------------------------->
             *|        |                             |     h1
             *|       \|/                            |
              | *          Zona B     |              |
              |    *                  |              |
              | Zona C  *             \|/             |
              |            *                         |         D punto deseado
```

```
                |---/|\-----------D            |         * variedad
                |   |                *        |
                |   |                       * |
                |   |     Zona A    /|\        *
                |   |                |         *
                |                              *
           h2 |-------------------------------+
              v

*/
   TLogico HayCambios=FALSO;
   Real h2,h1,ti,tip,h1p,h2p;
   char i2,i1,iti,i1p,itip;
   LargoSS PtosEta,PtosA,NoProA,PtosB,NoProB,PtosC,NoProC;
   TNodo *pet2,*pet1,*p2p,*pact,*p1p,*ppre;
   char *pv;
   Real Imin,F,ih1min,itimin,Taumin,tau,qlmin,ql,qcmin,qc;

   /*Constantes utiles*/
   Real h0=Par.h0;
   Real d1=H1MAX/Par.Nph1;         /*discretizacion en h1*/
   Real d2=H2MAX/Par.Nph2;   /*discretizacion en h2*/
   Real dTi=Par.dTi;
   SinSigno tcolTi=Par.NpTi+1;   /*elementos en coluna de tita*/
   SinSigno tfilh1=Par.Nph1+1;        /*elementos en una fila de h1*/
   SinSigno tplano=tcolTi*tfilh1; /*tamanio de un plano*/
   char nph2=Par.Nph2,nph1=Par.Nph1,npti=Par.NpTi;
   /*cotas*/
   Real qlmax=Par.qlMAX; Real qcmax=Par.qcMAX;

   /*Zona A*/
   for(i2=EtaDes+1, pet2=Reg+(int)tplano*i2, h2=i2*d2, pv=Var+i2,
               p2p=pet2-(int)tplano, h2p=h2-d2, PtosA=NoProA=0;
       i2<=nph2;
       i2++, p2p=pet2, h2p=h2, pet2+=(int)tplano, h2+=d2, pv++)
   {
      for(i1=*pv, pet1=pet2+tcolTi*i1, h1=i1*d1, PtosEta=0;
          i1>=0;
          i1--, pet1-=tcolTi, h1-=d1)
         for(iti=0, pact=pet1+iti, ti=iti*dTi;
             iti<=npti;
             iti++, pact++, ti+=dTi, PtosA++)
      /*Tenemos situado el punto actual*/
      {
            for(i1p=*(pv-1), p1p=p2p+tcolTi*i1p, h1p=i1p*d1,
                  Imin=pact->I;
                i1p>=0;
                i1p--, p1p-=tcolTi, h1p-=d1)
               for(itip=0, ppre=p1p+itip, tip=itip*dTi;
                   itip<=npti;
                   itip++, ppre++, tip+=dTi)
         /*Ya tenemos los dos puntos.*/
         if(ppre->h1sig>-1)  /*Solo si el sig.  es permitido*/
         {
            /*Calculo de los comandos para hacer la transicion*/
            /*tiempo no negativo*/
            if((tau=TAU(h1,h2,ti,h1p,h2p,tip,ppre))<0)
              break; /*ya que tau no depende de tip continue; */
            /*Flujo liquido < la cota y > 0*/
            if((ql=QL(h1,h2,ti,h1p,h2p,tip,tau,ppre))>qlmax || ql<0)
               break; /*ya que ql no depende de tip continue; */
            /*Flujo calor  < la cota y > 0*/
            if((qc=QC(h1,h2,ti,h1p,h2p,tip,tau,ppre))>qcmax || qc<0)
               continue;
            /*Se trata de paso valido, Pasamos a calcular su costo*/
            F=COSTO(h1,ti,ql,qc,tau,Par)+ppre->I;
            if(F<Imin)
            {
               Imin=F; ih1min=i1p; itimin=itip;
               Taumin=tau; qlmin=ql; qcmin=qc;
            }
         }
         if(Imin<pact->I)  /*para funcionar en varias pasadas*/
         {
            pact->I=Imin; pact->h1sig=ih1min; pact->tisig=itimin;
            pact->Tau=Taumin; pact->ql=qlmin; pact->qc=qcmin;
      pact->noCambios++; HayCambios=VERDAD;
            NoProA++; PtosEta++;
         }
       }
      printf("\nEtapa i2=%d\t\tNo. de puntos validos=%lu",i2,PtosEta);
   }
/*Fin de zona A*/
   printf("\nTerminada zona A.");
   printf("\nNo. puntos no prohibidos=%lu de %lu",NoProA,PtosA);

   /*Zona B*/
   for(i2=nph2-1, pet2=Reg+(int)tplano*i2, h2=i2*d2, pv=Var+i2,
               p2p=pet2+(int)tplano, h2p=h2+d2, PtosB=NoProB=0;
       i2>=0;
       i2--, p2p=pet2, h2p=h2, pet2-=(int)tplano, h2-=d2, pv--)
   {
      for(i1=*pv+1, pet1=pet2+tcolTi*i1, h1=i1*d1, PtosEta=0;
          i1<=nph1;
          i1++, pet1+=tcolTi, h1+=d1)
         for(iti=0, pact=pet1+iti, ti=iti*dTi;
             iti<=npti;
             iti++, pact++, ti+=dTi, PtosB++)
      /*Tenemos situado el punto actual*/
      {
            for(i1p=(*(pv+1)>=0?*(pv+1):0), p1p=p2p+tcolTi*i1p, h1p=i1p*d1,
                  Imin=pact->I;
                i1p<=nph1;
                i1p++, p1p+=tcolTi, h1p+=d1)
               for(itip=0, ppre=p1p+itip, tip=itip*dTi;
                   itip<=npti;
                   itip++, ppre++, tip+=dTi)
         /*Ya tenemos los dos puntos.*/
         if(ppre->h1sig>-1)  /*Solo si el sig.  es permitido*/
         {
            /*Calculo de los comandos para hacer la transicion*/
            /*tiempo no negativo*/
            if((tau=TAU(h1,h2,ti,h1p,h2p,tip,ppre))<0)
              break; /*continue;*/
            /*Flujo liquido < la cota y > 0*/
            if((ql=QL(h1,h2,ti,h1p,h2p,tip,tau,ppre))>qlmax || ql<0)
```

```
                  break; /*continue;*/
               /*Flujo calor  < la cota y > 0*/
               if((qc=QC(h1,h2,ti,h1p,h2p,tip,tau,ppre))>qcmax || qc<0)
                  continue;
               /*Se trata de paso valido, Pasamos a calcular su costo*/
               F=COSTO(h1,ti,ql,qc,tau,Par)+ppre->I;
               if(F<Imin)
               {
                  Imin=F; ih1min=i1p; itimin=itip;
                  Taumin=tau; qlmin=ql; qcmin=qc;
               }
            }
            if(Imin<pact->I)
            {
               pact->I=Imin; pact->h1sig=ih1min; pact->tisig=itimin;
               pact->Tau=Taumin; pact->ql=qlmin; pact->qc=qcmin;
      pact->noCambios++; HayCambios=VERDAD;
               NoProB++; PtosEta++;
            }
         }  /*Fin de zona B*/
         printf("\nEtapa i2=%d\t\tNo. de puntos validos=%lu",i2,PtosEta);
      }
      printf("\nTerminada zona B.");
      printf("\nNo. puntos no prohibidos=%lu de %lu",NoProB,PtosB);

      /*Zona C*/
      for(i2=1, pet2=Reg+(int)tplano*i2, h2=i2*d2, pv=Var+i2,
               p2p=pet2-(int)tplano, h2p=h2-d2, PtosC=NoProC=0;
          i2<=EtaDes;
          i2++, p2p=pet2, h2p=h2, pet2+=(int)tplano, h2+=d2, pv++)
      {
         for(i1=(i2==EtaDes)?*pv-1:*pv, pet1=pet2+tcolTi*i1, h1=i1*d1,
             PtosEta=0; i1>=0;
             i1--, pet1-=tcolTi, h1-=d1)
            for(iti=0, pact=pet1+iti, ti=iti*dTi;
                iti<=npti;
                iti++, pact++, ti+=dTi, PtosC++)
         /*Tenemos situado el punto actual*/
         {
               for(i1p=*(pv-1), p1p=p2p+tcolTi*i1p, h1p=i1p*d1,
                     Imin=pact->I;
                   i1p>=0;
                   i1p--, p1p-=tcolTi, h1p-=d1)
                  for(itip=0, ppre=p1p+itip, tip=itip*dTi;
                      itip<=npti;
                      itip++, ppre++, tip+=dTi)
               /*Ya tenemos los dos puntos.*/
               if(ppre->h1sig>-1)  /*Solo si el sig.  es permitido*/
               {
                  /*Calculo de los comandos para hacer la transicion*/
                  /*tiempo no negativo*/
                  if((tau=TAU(h1,h2,ti,h1p,h2p,tip,ppre))<0)
                     break; /*continue;*/
                  /*Flujo liquido < la cota y > 0*/
                  if((ql=QL(h1,h2,ti,h1p,h2p,tip,tau,ppre))>qlmax || ql<0)
                     break; /*continue;*/
                  /*Flujo calor  < la cota y > 0*/
                  if((qc=QC(h1,h2,ti,h1p,h2p,tip,tau,ppre))>qcmax || qc<0)
                     continue;
                  /*Se trata de paso valido, Pasamos a calcular su costo*/
                  F=COSTO(h1,ti,ql,qc,tau,Par)+ppre->I;
                  if(F<Imin)
                  {
                     Imin=F; ih1min=i1p; itimin=itip;
                     Taumin=tau; qlmin=ql; qcmin=qc;
                  }
               }
               if(Imin<pact->I)
               {
                  pact->I=Imin; pact->h1sig=ih1min; pact->tisig=itimin;
                  pact->Tau=Taumin; pact->ql=qlmin; pact->qc=qcmin;
      pact->noCambios++; HayCambios=VERDAD;
               NoProC++; PtosEta++;
            }
         }  /*Fin de zona C*/
         printf("\nEtapa i2=%d\t\tNo. de puntos validos=%lu",i2,PtosEta);
      }
      printf("\nTerminada zona C.");
      printf("\nNo. puntos no prohibidos=%lu de %lu",NoProC,PtosC);
      Par.NoProhi=NoProC+NoProB+NoProA;
      printf("\nNo. puntos no prohibidos totales=%lu de %lu(%lu)",
          Par.NoProhi,PtosA+PtosB+PtosC,Par.NumNodos);
      return HayCambios;
}
```

A.1.3.3 crea3d.c

```
#include <stdlib.h>
#include <stdio.h>

#include "din3d.h"

int CreaReg(par,reg,var,etapDes)
TParametros par;
TNodo **reg;
char **var,*etapDes;
{
   /*funcion que crea la regilla:
     - pide la memoria para reg
     - inicializa los valores
     - crea var con los datos de la variedad
     - pone el punto final
*/
   LargoSS i,ih1d,iti;
   TNodo *na,*pa;
   Real h1,h2;
   char *ptv;

   /*Calculo del la memoria necesaria*/
      /*Ordenacion  -por tita (mas rapido)
                    -por h1
                    -por h2 (menos rapido) */
```

```
   if(!(*reg=malloc(par.NumNodos*sizeof(TNodo))))
   {
      printf("\nNo se obtuvo memoria para la regilla");
      return 0;
   }

   /*memoria para var*/
   if(!(*var=malloc((par.Nph2+1)*sizeof(char))))
   {
      printf("\nNo se obtuvo memoria para var");
      return 0;
   }

   /*inicializacion de valores*/
   for(i=1, na=*reg; i<=par.NumNodos; i++, na++)
   {
      na->I=HUGE_VAL;
      na->h1sig=na->tisig=-1;
      na->noCambios=0;
   }

   /*determinamos los puntos de la variedad*/
   for(i=0, h2=0.0, ptv=*var; i<=par.Nph2;
      i++, h2+=(H2MAX/(Real)par.Nph2), ptv++)
   {
      h1=(R2/R1+1)*h2-par.h0;
      if(h1>H1MAX) *ptv=par.Nph1;  /*Nos pasamos  por la decha*/
      else if (h1<0.0) *ptv=-1;       /*Nos pasamos en uno por la izda*/
      else *ptv=ROUND(h1*par.Nph1/H1MAX);
   }

#ifdef VAR_FINAL
   /*Inicializacion de la variedad a un valor distinto*/
   for(i=0, ptv=*var, pa=*reg; i<=par.Nph2;
       i++, ptv++, pa+=(par.Nph1+1)*(par.NpTi+1))
      for(iti=0, na=pa+*ptv*(par.NpTi+1); iti<=par.NpTi; iti++, na++)
      {
         na->h1sig=na->tisig=127;
         na->I=1e6;     /*Coste no comparable con el pto final*/
         na->ql=na->qc=na->Tau=0.0;
      }
#endif

   /*Volvemos a comprobar la validez de la altura deseada*/
   if((h1=(R1/R2+1)*par.h2des-par.h0)<0 || h1>H1MAX)
   {
      fprintf(stderr,"\nAltura h2 deseada inalcanzable");
      exit(1);
   }

   /*fijamos punto final haciendo 0 sus variables siguientes*/
   *etapDes=ROUND(par.h2des*par.Nph2/H2MAX);
   i=ROUND(par.Tides/par.dTi);   /*indice del Tita deseado*/
   ih1d=*(*var+*etapDes);
   na=*reg;
   na+=((*etapDes)*(par.Nph1+1)+ih1d)*(par.NpTi+1)+i;
   na->h1sig=na->tisig=0;
   na->ql=R2*par.h2des;  /*valor estacionario para el flujo*/
   /*na->ql=0;*/
   na->qc=par.Tides/RT;  /*valor estacionario para el calor*/
   /*na->qc=0;*/
   /*na->Tau=???? */
   na->Tau=0;    /*0 da problemas, 1 por probar*/
     /*Podemos tomar 0 porque hacemos la aroximacion*/
   na->I=0;     /*unico punto de costo 0*/

   return 1;
}
```

A.1.3.4 io3d.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "din3d.h"

int LeePar(par,NomFP,Todo)
TParametros *par;
char *NomFP;
TLogico Todo;
{
   /*Rutina que lee los parametros, de fichero o interactivamente*/
   FILE *fp;
   char c;
   int  tmp;  Real tmpR,h1d;

   if(*NomFP)
   {
      /*existe nombre de fichero -> lo intentamos abrir */
      if(!(fp=fopen(NomFP,"r")))
      {
     printf("\nNO se pudo abrir fichero de parametros");
     return 0;
      }
      fscanf(fp,"%lg",&par->h0);
      fscanf(fp,"%d",&tmp); par->Nph1=tmp;
      fscanf(fp,"%d",&tmp); par->Nph2=tmp;
      fscanf(fp,"%d",&tmp); par->NpTi=tmp;
      fscanf(fp,"%lg",&par->dTi);
      fscanf(fp,"%lg",&par->qlMAX);
      fscanf(fp,"%lg",&par->qcMAX);
      fscanf(fp,"%lg",&par->FacT);
      fscanf(fp,"%lg",&par->FacQl);
      fscanf(fp,"%lg",&par->FacQc);
      fscanf(fp,"%lg",&par->FacH1);
      fscanf(fp,"%lg",&par->FacTi);
      fscanf(fp,"%lg",&par->h2des);
      fscanf(fp,"%lg",&par->Tides);
      par->h2des=ROUND(par->h2des*par->Nph2/H2MAX)*H2MAX/par->Nph2;
      par->Tides=ROUND(par->Tides/par->dTi)*par->dTi;
      par->NumNodos=(par->Nph1+1)*(par->Nph2+1)*(par->NpTi+1);
      if((h1d=(R1/R2+1)*par->h2des-par->h0)<0 || h1d>H1MAX)
      {
```

```
            fprintf(stderr,"\nAltura h2 deseada inalcanzable");
            exit(1);
         }
         }
         return 1;
      }

      /*menu para la mdificacion de los parametros*/
      do {
         if(!Todo)
         {
      printf("\n\n\n1-Diferencia de altura h0=\t%lg",par->h0);
      printf("\n\n2-Numero de puntos dimension h1=\t%d",par->Nph1);
      printf("\n3-Numero de puntos dimension h2=\t%d",par->Nph2);
      printf("\n4-Numero de puntos dimension Tita=\t%d",par->NpTi);
      printf("\n5-Discretizacion en tita=\t%lg",par->dTi);
      printf("\n\n6-Maximo caudal liquido=\t%lg",par->qlMAX);
      printf("\n7-Maximo caudal calorifico=\t%lg",par->qcMAX);
      printf("\n\nt-Factor tiempo en costo=\t%lg",par->FacT);
      printf("\nl-Factor caudal liquido en costo=\t%lg",par->FacQl);
      printf("\nc-Factor caudal calor en costo=\t%lg",par->FacQc);
      printf("\nh-Factor h1 en costo=\t%lg",par->FacH1);
      printf("\ni-Factor Tita en costo=\t%lg",par->FacTi);
      printf("\n\nf-altura 2 pto final=\t%lg",par->h2des);
      printf("\ng-Tita pto final=\t%lg",par->Tides);
      printf("\n\nPulsa inicial del parametro a modificar.(0 salir)");
      while((c=getchar())<' ');
         }
         else c='1';
         switch (c|0x20)
         {
         case '1':
      printf("\nDiferencia de altura h0=\t");
      scanf("%lg",&par->h0); if(!Todo) break;
         case '2':
      do {
               printf("\nNumero de puntos dimension h1=\t");
         scanf("%d",&tmp);
      } while(tmp<=0 || tmp>=128); par->Nph1=tmp; if(!Todo) break;
         case '3':
      do {
         printf("\nNumero de puntos dimension h2=\t");
         scanf("%d",&tmp);
      } while(tmp<=0 || tmp>=128); par->Nph2=tmp;
      par->h2des=ROUND(par->h2des*par->Nph2/H2MAX)*H2MAX/par->Nph2;
      if(!Todo) break;
         case '4':
      do {
         printf("\nNumero de puntos dimension Tita=\t");
         scanf("%d",&tmp);
      } while(tmp<=0 || tmp>=128); par->NpTi=tmp; if(!Todo) break;
         case '5':
            printf("\nDiscretizacion en tita=\t");
      scanf("%lg",&par->dTi);
      par->Tides=ROUND(par->Tides/par->dTi)*par->dTi;
      if(!Todo) break;
         case '6':
            printf("\nMaximo caudal liquido=\t");
      scanf("%lg",&par->qlMAX); if(!Todo) break;
         case '7':
            printf("\nMaximo caudal calorifico=\t");
      scanf("%lg",&par->qcMAX); if(!Todo) break;
         case 't':
            printf("\nFactor tiempo en costo=\t");
      scanf("%lg",&par->FacT); if(!Todo) break;
         case 'l':
            printf("\nFactor caudal liquido en costo=\t");
      scanf("%lg",&par->FacQl); if(!Todo) break;
         case 'c':
            printf("\nFactor caudal calor en costo=\t");
      scanf("%lg",&par->FacQc); if(!Todo) break;
         case 'h':
            printf("\nFactor h1 en costo=\t");
      scanf("%lg",&par->FacH1); if(!Todo) break;
         case 'i':
            printf("\nFactor Tita en costo=\t");
      scanf("%lg",&par->FacTi); if(!Todo) break;
         case 'f':
            do {
               printf("\naltura 2 del pto final=\t");
               scanf("%lg",&tmpR);
      } while ((tmpR<0.0) || (tmpR>H2MAX));
      if(par->Nph2)
         par->h2des=ROUND(tmpR*par->Nph2/H2MAX)*H2MAX/par->Nph2;
      else
         par->h2des=0;
      if(!Todo) break;
         case 'g':
            do {
               printf("\nTita pto final=\t");
               scanf("%lg",&tmpR);
      } while((tmpR<0.0) || (tmpR>(par->NpTi*par->dTi)));
      if(par->dTi)
         par->Tides=ROUND(tmpR/par->dTi)*par->dTi;
      else par->Tides=0.0;
      if(!Todo) break;
         }
         Todo=FALSO;
         if(c=='0')  /*Comprobamos que sea h2 posible*/
            if((h1d=(R1/R2+1)*par->h2des-par->h0)<0 || h1d>H1MAX)
            {
               printf("\nAltura h2 deseada inalcanzable.");
               printf("\nDebes modificar h2 deseada o h0");
               c=1;
            }
      } while(c!='0');
      par->NumNodos=(par->Nph1+1)*(par->Nph2+1)*(par->NpTi+1);
      return 1;
}

int SalvaReg(Par,Reg,NFil)
TParametros Par;
TNodo *Reg;
char *NFil;
{
   /*Rutina para salvar la regilla*/
```

```
   FILE *fr;
   char NFtmp[30],VerNum=VERNUM;

   if(*NFil)     /*Existe nombre*/
   {
      if(!(fr=fopen(NFil,"wb")))
      {
   /*No se pudo abrir el fichero por defecto vamos a tmp*/
   strcpy(NFtmp,"/tmp/");
   strcat(NFtmp,NFil);
   if(!(fr=fopen(NFtmp,"wb")))
   {
      printf("\nNo se pudo abrir ni el fichero de emergencia");
      return 0;
   }
      }
   }
   else       /*Preguntamos el nombre*/
   {
      do {
   *NFtmp=0;
   printf("\nNombre del fichero:");
   scanf("%s",NFtmp);
       } while (*NFtmp && !(fr=fopen(NFtmp,"wb")));
       if(!*NFtmp)
       {
   printf("\nNo se especifica nombre: no se salva datos");
   return 0;
       }
       if(!(fr=fopen(NFtmp,"wb")))
       {
          printf("\nNo se pudo abrir ni el fichero de emergencia");
          return 0;
       }
   }

   /*Comenzamos a salvar*/
   /*primero el numero de la version*/
   fwrite(&VerNum,sizeof(char),1,fr);

   fwrite(&Par,sizeof(TParametros),1,fr);
   fwrite(Reg,sizeof(TNodo),Par.NumNodos,fr);

   fclose(fr);
   return 1;
}
```

A.1.3.5 prin3d.c

```
#include <stdlib.h>
#include <stdio.h>

#include "din3d.h"

TLogico Calcula();

main(argc,argv)
int argc;
char *argv[];
{
   TParametros Par;
   TNodo *Regilla;
   char *Variedad,EtapaDes,pasada;
   TLogico HayCambio;

   char NFPar[20],NFReg[20];
   *NFPar=*NFReg='\0';
   printf("Numero de argumentos %d",argc);

   /*Cambiamos para que el primer paramatro sea fichero de datos*/
   if(argc>2)
   {
      strcpy(NFPar,argv[2]);
      LeePar(&Par,NFPar,VERDAD);
      printf("Estan leidos los parametros\n");
      *NFPar=0;
      LeePar(&Par,NFPar,FALSO);
   }
   else
      LeePar(&Par,NFPar,VERDAD);
   CreaReg(Par,&Regilla,&Variedad,&EtapaDes);
   for(pasada=1, HayCambio=VERDAD; HayCambio; pasada++)
   {
      printf("\n\n Pasada %d", pasada);
      HayCambio=Calcula(Par,Regilla,Variedad,EtapaDes);
   }
   printf("\n Numero total de pasadas %d",pasada);
   if(argc>1)  strcpy(NFReg,argv[1]);
   SalvaReg(Par,Regilla,NFReg); /*para que pida todos los par*/

   return 0;
}
```

A.1.3.6 sacmat3d.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "din3d.h"

/*Programa para sacar informacion a matlab de fichero dat*/


void Procesa(fd,fm)
FILE *fd,*fm;
{
   char version;
   TParametros Par;
```

```
   CabezaMat Cab;
   char *NomInd="Indices",*NomPar="Param";
   Real TemReal;
   fpos_t posDat;
   LargoSS i;
   TNodo na;

   /*funcion que leera el fichero de datos he ira creando el mat*/
   /*Primero leemos el numero de version y parametros*/
   fread(&version,sizeof(char),1,fd);
   fread(&Par,sizeof(TParametros),1,fd);

   /*creamos cabecera de  matriz*/
   Cab.tipo=1000;
   Cab.HayImag=0;

   /*Salvamos matriz con parametros*/
   Cab.NumFilas=1;
   Cab.NumColum=16;  /*numero total de parametros*/
   Cab.TamNom=strlen(NomPar)+1;
   fwrite(&Cab,sizeof(CabezaMat),1,fm);
   fwrite(NomPar,sizeof(char),Cab.TamNom,fm);
   fwrite(&Par.h0,sizeof(Real),3,fm);
   TemReal=Par.Nph1; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.Nph2; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.NpTi; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.NumNodos; fwrite(&TemReal,sizeof(Real),1,fm);
   TemReal=Par.NoProhi; fwrite(&TemReal,sizeof(Real),1,fm);
   fwrite(&Par.dTi,sizeof(Real),8,fm);
   /*ya esta la matriz de parametros*/

   fgetpos(fd,&posDat);  /*tomamos nota del principio de los datos*/

   Cab.NumFilas=Par.NumNodos;
   Cab.NumColum=1;
   Cab.TamNom=strlen(NomInd)+1;

   fwrite(&Cab,sizeof(CabezaMat),1,fm);
   fwrite(NomInd,sizeof(char),Cab.TamNom,fm);
   for(i=1; i<=Par.NumNodos; i++)
   {
      fread(&na,sizeof(TNodo),1,fd);
      TemReal=na.h1sig+na.tisig*1000+na.noCambios*1000000;
      fwrite(&TemReal,sizeof(Real),1,fm);
   }

#ifdef PRUEBA
   /*Volvemos al punto inicial*/
   fsetpos(fd,&posDat);

   Cab.TamNom=strlen(NomTau)+1;
   fwrite(&Cab,sizeof(CabezaMat),1,fm);
   fwrite(NomInd,sizeof(char),Cab.TamNom,fm);

   for(i=1; i<=Par.NumNodos; i++)
   {
      fread(&na,sizeof(TNodo),1,fd);
      IndMasCam=na.h1sig+na.tisig*1000+na.noCambios*1000000;
      fwrite(IndMasCam,sizeof(Real),1,fm);
   }
#endif


}
main(argc,argv)
int argc;
char *argv[];
{
   FILE *fd,*fm;

   if((argc<1) || !(fd=fopen(argv[1],"rb")))
   {
      printf("\n Necesario especificar fichero de datos");
      exit(1);
   }
   if((argc<2) || !(fm=fopen(argv[2],"wb")))
   {
      printf("\n No se puede abrir el fichero matlab");
      exit(2);
   }
   Procesa(fd,fm);

   fclose(fd);
   fclose(fm);
   return 0;
}
```

A.1.3.7 tray3d.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "din3d.h"

/*Programa para sacar informacion a matlab de fichero dat*/
#define MAXPTINI 20

int  LeeDatos(fd,par,reg)
FILE *fd;
TParametros *par;
TNodo **reg;
{
   char version;

   /*funcion que leera el fichero de datos */
   /*Primero leemos el numero de version y parametros*/
   fread(&version,sizeof(char),1,fd);
   fread(par,sizeof(TParametros),1,fd);

   /*Pedimos memoria*/
   if(!(*reg=malloc(par->NumNodos*sizeof(TNodo))))
```

```
   {
      printf("\nNo se pudo obtener memoria para regilla");
      return 0;
   }
   return (fread(*reg,sizeof(TNodo),par->NumNodos,fd))==par->NumNodos;

}

char LeePtIni(NomIni,par,ptini)
char *NomIni;
TParametros par;
Real *ptini;
{
   /*funcion que lee los puntos iniciales*/
   Real *pt;
   char k;
   FILE *fi;

   if(*NomIni)  /*se ha especificado nombre*/
   {
      if(!(fi=fopen(NomIni,"r")))
      {
  fprintf(stderr,"\nNo se puede abrir fichero de iniciales");
  return 0;
      }
      for(pt=ptini, k=0; k<=3*MAXPTINI &&
  (fscanf(fi,"%lg",pt)!=EOF); k++, pt++);
      return k/3;
   }
   printf("\nEntra los puntos iniciales(hasta %u)",MAXPTINI);
   printf("\nEntra un carcter para terminar");
   for(k=0, pt=ptini; k<MAXPTINI; k++)
   {
      printf("\n Punto inicial %u:\n",k+1);
      printf("h1=");
      if(!scanf("%lf",pt)) break;
      pt++;
      printf("h2=");
      if(!scanf("%lf",pt)) break;
      pt++;
      printf("ti=");
      if(!scanf("%lf",pt)) break;
      pt++;
   } /* del while k */
   return k;
}


void CalSal(par,reg,ptini,Nptos,fm)
TParametros par;
TNodo *reg;
Real *ptini;
char  Nptos;
FILE *fm;
{
#define DIRNODO(ih1,ih2,iti)  (reg+((ih2)*(par.Nph1+1)+(ih1))*\
   (par.NpTi+1)+(iti))
   /*Funcion que calcula y salva trayectorias*/
   CabezaMat Cab;
   char *NomTra="trayec";
   Real *pt,w[6],Costo;  /*vector de la anchura de matriz*/
   char ih1,ih2,iti,ih2des,p,i,nptos,Sentido;
   SinSigno Numfil;
   TNodo *na;


 /*creamos cabecera de  matriz*/
   Cab.tipo=1100;
   Cab.HayImag=0;

   /*Salvamos matriz con parametros*/
   Cab.NumFilas=1;
   Cab.NumColum=2*3+1;  /*numero total de parametros*/
   Cab.TamNom=strlen(NomTra)+1;
   /*Salvamos Cabeza incompleta*/
   fwrite(&Cab,sizeof(CabezaMat),1,fm);
   fwrite(NomTra,sizeof(char),Cab.TamNom,fm);

   ih2des=ROUND(par.h2des*par.Nph2/H2MAX);
   for(i=1, pt=ptini, Numfil=0; i<=Nptos; i++, pt+=3)
   {
      if((pt[0]>H1MAX)||(pt[1]>H2MAX)) /*h1o h2 mayores que maximo*/
  continue;  /*pasamos de ese punto*/
      ih1=ROUND(pt[0]*par.Nph1/H1MAX);
      ih2=ROUND(pt[1]*par.Nph2/H2MAX);
      iti=ROUND(pt[2]/par.dTi);
      if(iti>par.NpTi) /*tita mayor del considerado*/
  continue; /*pasamos de ese punto*/

      na=DIRNODO(ih1,ih2,iti);
      if(na->h1sig!=-1)
         if (ih2>ih2des)
  {
     nptos=ih2-ih2des+1; Sentido=-1;
  }
  else
  {
     nptos=ih2des-ih2+1; Sentido=1;
  }
      else
  nptos=1;

      /*debemos salvar 3 filas iniciales de la trayectoria*/
      w[0]=nptos; w[1]=3; w[2]=3; w[3]=na->I;
      fwrite((void *)w,sizeof(Real),7,fm); Numfil++;
      w[0]=par.h0; w[1]=par.h2des; w[2]=par.Tides;
      fwrite((void *)w,sizeof(Real),7,fm); Numfil++;
      w[0]=par.FacT; w[1]=par.FacQl; w[2]=par.FacQc;
      w[3]=par.FacH1; w[4]=par.FacTi;
      fwrite((void *)w,sizeof(Real),7,fm); Numfil++;

      /*Comenzamos a salvar los puntos*/
      for(p=1, Costo=0 ;p<=nptos; p++)
      {
  w[0]=ih1*H1MAX/par.Nph1;  /*h1 pto actual*/
  w[1]=ih2*H2MAX/par.Nph2;  /*h2 pto actual*/
```

```
   w[2]=iti*par.dTi;
   w[3]=na->q1; w[4]=na->qc; w[5]=na->Tau;
   w[6]=Costo;
         fwrite((void *)w,sizeof(Real),7,fm); Numfil++;
   /*Pasamos al nodo siguiente*/
   Costo+=COSTO(w[0],w[2],w[3],w[4],w[5],par);
   if((na->h1sig==-1) && (p>1))
      exit(3);
   ih2+=Sentido; ih1=na->h1sig; iti=na->tisig;
   na=DIRNODO(ih1,ih2,iti);
      }
   } /*bucle de ptos iniciales*/

   rewind(fm); /*volvemos al principio para corregir la cabeza*/
   if(fseek (fm, 0L, SEEK_SET))
   {
      fprintf(stderr,"\nProblema al volver al principio\n");
      return;
   }

   Cab.NumFilas=Numfil;
   printf("\nNumero de filas %d\n",Cab.NumFilas);
   /*Salvamos Cabeza completa*/
   if(!fwrite(&Cab,sizeof(CabezaMat),1,fm))
   {
      fprintf(stderr,"\nProblema al machacar cabeza\n");
   }
}

main(argc,argv)
int argc;
char *argv[];
{
   FILE *fd,*fm;
   TNodo *reg;
   TParametros par;
   char NomIni[20];
   SinSigno Nptos;
   Real ptini[MAXPTINI*3];

   if((argc<1) || !(fd=fopen(argv[1],"rb")))
   {
      fprintf(stderr,"\n Necesario especificar fichero de datos\n");
      exit(1);
   }
   if((argc<2) || !(fm=fopen(argv[2],"wb")))
   {
      fprintf(stderr,"\n No se puede abrir el fichero matlab\n");
      exit(2);
   }
   *NomIni=0;
   if(argc>=3)
      strcpy(NomIni,argv[3]);
   LeeDatos(fd,&par,&reg);
   Nptos=LeePtIni(NomIni,par,ptini);
   CalSal(par,reg,ptini,Nptos,fm);


   fclose(fd);
   fclose(fm);
   return 0;
}
```

A.2 Redes Neuronales

A.2.1 Código Común

A.2.1.1 defred.h

```
#include <stdlib.h>
#include <stdio.h>

/* ************************************************ */
#define VERNUM  0xA001  /*Se aÑade en red elnumero de gradientes
                          y trayectoria deseada a partir de 0xA001*/
/* ************************************************ */


#define NDAT 11
#define  VE  3  /*ahora tenemos 3 variables de estado:
                  - (X1,X2)  Posicion
                  - X3  Orientacion respecto a X1 */
#define  VC  3  /*Tenemos 2 comandos:
                  - Avance segun orientacio
                  - Cambio de orientacion
                  - Intervalo temporal*/
#define MAXPTINI 20

/************ GENERAL *********************************/

#include "tdatos.h"
```

A.2.1.2 funherra.h

```
/*****************************************************************
********** FUNCIONES HERRAMIENTA ********************************
*****************************************************************/
/* como macros ********/
Real G(tipo,z)
char tipo;
```

```
Real z;
{
   switch (tipo)
   {
   case TH: /*tangente hiperbolica*/
      return tanh(z);
   case SG:  /*sigmoide*/
      return 1/(1+exp(-z));
   case LI:  /*lineal*/
      return z;
   }
   exit(3);  /*acceso incorrecto*/
}

Real DerivadaG(tipo,y)
char tipo;
Real y;
{
   switch (tipo)
   {
   case TH:  /*tangente hiperbolica*/
      return (1-(y)*(y));
   case SG:  /*sigmoide*/
      return   y*(1-y);
   case LI:  /*lineal*/
      return 1;
   }
   exit(3);  /*acceso incorecto*/
}

void SumaX(X1,X2,Resul)
Real X1[VE],X2[VE],Resul[VE];
{
   SinSigno i;
   Real *p1,*p2,*pr;

   for(i=1, p1=X1, p2=X2, pr=Resul; i<=VE; i++, p1++, p2++, pr++)
      *pr=(*p1)+(*p2);
}


void SumaU(U1,U2,Resul)
Real U1[VC],U2[VC],Resul[VC];
{
   SinSigno i;
   Real *p1,*p2,*pr;
   for(i=1, p1=U1, p2=U2, pr=Resul; i<=VC; i++, p1++, p2++, pr++)
      *pr=(*p1)+(*p2);
}

Real ProdEscX(X1,X2)
Real X1[VE],X2[VE];
{
   SinSigno i;
   Real *p1,*p2,Resul;
   Resul=0.0;
   for(i=1, p1=X1, p2=X2; i<=VE; i++, p1++, p2++)
      Resul+=(*p1)*(*p2);
   return Resul;
}

Real ProdEscC(U1,U2)
Real U1[VC],U2[VC];
{
   SinSigno i;
   Real *p1,*p2,Resul;
   Resul=0.0;
   for(i=1, p1=U1, p2=U2; i<=VC; i++, p1++, p2++)
      Resul+=(*p1)*(*p2);
   return Resul;
}

void CalculaAlfa(PRed)
TRed *PRed;
{
   /*por ahora alfa cte*/
   if(PRed->AlgBus & BSimple)
      PRed->Alfa=PRed->c3;
   else
      PRed->Alfa=PRed->c1*PRed->NumItera
         +PRed->c3/(1+PRed->c2*PRed->NumItera);
}

#ifdef __hp9000s300
   int isinf(Num)
   Real Num;
   {
      return Num>=HUGE_VAL;
   }
#endif
```

A.2.1.3 caldel1.h

```
void CalcDelta(PUnidad,EstDese,NumEta)
TUnidad *PUnidad;
Real *EstDese;
SinSigno NumEta;
{
   /* Dado:
   el Landa de la etapa sigiente, aplica la
      Back-Propagation a esta unidad
   TODO EL PROCESO SE HARA DE ATRAS  ALANTE*/
   Real *LandaSig=(PUnidad+1)->landa;


   SinSigno ns,nsM1,p,q,s;
   Real *ds, /*apuntara al delta de na neurona de la etapa siguiente*/
        *ws; /*apuntara peso de neu. de capa siguiente*/
   Real  /*declaracion e inicializacion principales punteros*/
    *da=PUnidad->Delta+(PUnidad->TotNeu-1), /*Comenzamos por la ultima*/
      /*da Apunta a delta actualmente calculandose*/
    *Ds=da, /*Apunta Ultima neurona capa anterior*/
    *ya=PUnidad->Y+(PUnidad->TotNeu+VE-1), /*ultima neurona*/
```

```
      /*ya Apunta al valor de salida Y de la neurona actual*/
     *Ps=PUnidad->Wa+(PUnidad->TotPes-1);
      /*Apunta a peso q de la ultima neurona de la capa siguiente*/
    Real VCaux1[VC],VCaux2[VC], /*Vectores auxiliares de comando*/
  VEaux1[VE],VEaux2[VE]; /*    "             "    de estado*/


   ParcialUdeH(PUnidad->Y,PUnidad->u,VCaux1);
   for(q=VC; /*ultima capa=>tantas neuronas */
       q>=1;                    /* como salidas*/
       q--, ya--, da--)
   {
      Real PE,c=PUnidad->RanSal[q-1]; /*Rango de salida de la neurona*/

      ParcialUqF(PUnidad->Y,PUnidad->u,q,VEaux1);
      PE=ProdEscX(VEaux1,LandaSig)+VCaux1[q-1];

      /*funcion activacion indicada en FunSAl*/
      *da=DerivadaG(PUnidad->FunSal[q-1],*ya/c)*c*PE;
      /*ultima capa se multiplica por el factor correspondiente*/
   }
   /* bucle principal de la ultima a la primera capa
       para calcula delta de capa anterior */
   for(s=PUnidad->L; /*no.+1 de la capa que se esta calculando*/
       s>=2; s-- )
      {
         ns=PUnidad->NeuC[s-1]; /*no. neuronas capa que se esta calc.*/
         nsM1=PUnidad->NeuC[s]; /*no. neu. capa siguiente*/
         /* bucle sobre las neuronas de la capa s*/
         for(q=ns ; q>=1 ; q--)
         {
         /*calculo de los delta de cada neurona*/
            Real sum=0.0;

            /* bucle sobre las entradas que alimenta->Capa siguiente*/
            for(p=nsM1, ds=Ds, ws=Ps;
                p>=1;
                p--, ds--, ws-=ns+1)
                   sum+=(*ds)*(*ws);

            /* La funcion de activacion es siempre th  */
            *da=DerivadaG(TH,*ya)*sum; /* no factor, capa interiores */
      da--; /*Siguiente delta*/
      ya--; /*Siguente salida*/
      Ps--; /*cambiamos a pesos con p=q*/
         }
   /*al terminar con capa debemos actualizar DS
      se actualiza con el valor en que quedo ds */
   Ds=ds;
   /*Para actualizar Ps debemos saltarnos el Bias
      lo actualizamos a ws+ns-1 a Ps-(ns+1)*(nsM1-1)-1 */
   Ps=ws+ns-1;
      } /* for de capas que calcula delta de anterior */
if (NumEta)    /* No lo hacemos si estamos en la 1a. unidad*/
{
/* Calculo de landa */

   /* Calculo de landa de la etapa actual */
   /* Termino landa*parcial xk de fk*/
   for(q=0; q<VE; q++)
   {
      ParcialXqF(PUnidad->Y,PUnidad->u,q+1,VEaux1);
      VEaux2[q]=ProdEscX(LandaSig,VEaux1);
   }
   ParcialXdeH(PUnidad->Y,PUnidad->u,EstDese,NumEta,VEaux1);
   SumaX(VEaux1,VEaux2,PUnidad->landa);

   ns=PUnidad->NeuC[1];
   for(p=0, Ps=PUnidad->Wa+1; /*No BIAS*/
       p<VE ;
       p++, Ps++ )
   {
      for(q=0, ws=Ps, ds=PUnidad->Delta,
    VEaux1[p]=0.0 ;
    q<ns ; q++, ws+=VE+1, ds++ )
  VEaux1[p]+=(*ws)*(*ds);
   }

   SumaX(VEaux1,PUnidad->landa,PUnidad->landa);

}/* fin de la seccion de calculos complejos */
}/* fin de  CalcDelta*/

void ObtieneDelta(PRed)
TRed *PRed;
{

   TUnidad *UActual;
   SinSigno i;

   /*Calculo de landa N*/
    UActual=PRed->PtUni+PRed->N;  /* en la ultima cabecera */
    ParcialXnHn(UActual->Y,UActual->landa);

   /*primero se calculan las deltas
      de todas las neuronas sin tocar los pesos*/
   for(i=PRed->N-1, UActual--;
  i>=1;
  i--, UActual--)
      CalcDelta(UActual,PRed->TraDese,i);
  /*Caso especial de la 1a. unidad*/
   CalcDelta(UActual,PRed->TraDese,0);
}
```

A.2.1.4 caldel2.h

```
void CalcDelta(PUnidad,EstDes,NumEta)
TUnidad *PUnidad;
Real *EstDes; /*Necesarios para pesar trayectoria en H*/
SinSigno NumEta;
{
   /* Dado:
   el Landa de la etapa sigiente, aplica la
```

```
      Back-Propagation a esta unidad
    TODO EL PROCESO SE HARA DE ATRAS  ALANTE*/
    Real *LandaSig=(PUnidad+1)->landa,
   *ParJxsig=(PUnidad+1)->parJx;


    SinSigno ns,nsM1,p,q,s;
    Real *ds, /*apuntara al delta de na neurona de la etapa siguiente*/
         *ws,/*apuntara peso de neu. de capa siguiente*/
  *ppj ;
    Real  /*declaracion e inicializacion principales punteros*/
     *da=PUnidad->Delta+(PUnidad->TotNeu-1),
     /*Comenzamos por la ultima*/
       /*da Apunta a delta actualmente calculandose*/
     *Ds=da, /*Apunta Ultima neurona capa anterior*/
     *ya=PUnidad->Y+(PUnidad->TotNeu+VE-1), /*ultima neurona*/
       /*ya Apunta al valor de salida Y de la neurona actual*/
     *Ps=PUnidad->Wa+(PUnidad->TotPes-1);
       /*Apunta a peso q de la ultima neurona de la capa siguiente*/


    for(q=VE, ppj=ParJxsig+VE-1; /*ultima capa=>tantas neuronas */
        q>=1;                    /* como salidas*/
        q--, ya--, da--, ppj--)
    {
       Real c=PUnidad->RanSal[q-1]; /*Rango de salida de la neurona*/
       /*Funcion de salida indicada en FunSal*/
       *da=DerivadaG(PUnidad->FunSal[q-1],*ya/c)*c*(*ppj);
       /*ultima capa se multiplica por el factor correspondiente*/
    }
    /* bucle principal de la ultima a la primera capa
        para calcula delta de capa anterior */
    for(s=PUnidad->L; /*no.+1 de la capa que se esta calculando*/
        s>=2; s-- )
       {
          ns=PUnidad->NeuC[s-1]; /*no. neuronas capa que se esta calc.*/
          nsM1=PUnidad->NeuC[s]; /*no. neu. capa siguiente*/
          /* bucle sobre las neuronas de la capa s*/
          for(q=ns ; q>=1 ; q--)
          {
          /*calculo de los delta de cada neurona*/
             Real sum=0.0;

             /* bucle sobre las entradas que alimenta->Capa siguiente*/
             for(p=nsM1, ds=Ds, ws=Ps;
                 p>=1;
                 p--, ds--, ws-=ns+1)
                    sum+=(*ds)*(*ws);

             /* La funcion de activacion es siempre th  */
             *da=DerivadaG(TH,*ya)*sum; /* no factor, capa interiores */
       da--; /*Siguiente delta*/
       ya--; /*Siguente salida*/
       Ps--; /*cambiamos a pesos con p=q*/
          }
    /*al terminar con capa debemos actualizar DS
       se actualiza con el valor en que quedo ds */
    Ds=ds;
    /*Para actualizar Ps debemos saltarnos el Bias
       lo actualizamos a ws+ns-1 ã Ps-(ns+1)*(nsM1-1)-1 */
    Ps=ws+ns-1;
       } /* for de capas que calcula delta de anterior */
if (NumEta)    /* No lo hacemos si estamos en la 1a. unidad*/
{
/* Calculo de landa y parcial x deJ de esta etapa */
    Real VCaux1[VC],VCaux2[VC], /*Vectores auxiliares de comando*/
   VEaux1[VE],VEaux2[VE], /*    "             "   de estado*/
   SumPJ[VE];    /*se iran acumulando sumandos de ParcialJx*/

    /* Calculo de landa de la etapa actual */
    /* Termino landa*parcial Uk de fk*parcial Xk de fk1 */
    for(q=0; q<VC; q++)
    {
       ParcialUqF(PUnidad->Y,(PUnidad+1)->Y,PUnidad->u,
       (PUnidad+1)->u,q+1,VCaux1);
       VCaux2[q]=ProdEscC(LandaSig,VCaux1);
    }
    ParcialUdeH((PUnidad-1)->Y,PUnidad->u,VCaux1);
    SumaU(VCaux1,VCaux2,PUnidad->landa);

    /* Calculo de la Parcial x de J */
    /* Primero calculo PJy= sumas d*w para 1a capa 1*/
    ns=PUnidad->NeuC[1];
    for(p=0, Ps=PUnidad->Wa+1; /*No BIAS*/
        p<VE ;
        p++, Ps++ )
    {
       for(q=0, ws=Ps, ds=PUnidad->Delta,
     VEaux1[p]=0.0 ;
     q<ns ; q++, ws+=VE+1, ds++ )
    VEaux1[p]+=(*ws)*(*ds);
    }

    /* Calculo Parcial X de H (PHx) */
    ParcialXdeH(PUnidad->Y,(PUnidad+1)->u,EstDes,NumEta,VEaux2);

    SumaX(VEaux1,VEaux2,SumPJ);

    /* Calculo de los terminos mas complejos */
       /* Bucle para realizar calculo de terminos 1xVC*VCxVE */
       /* Termino landa*parcial Xk de f */
    for(p=0; p<VE; p++)
    {
       ParcialXqF(PUnidad->Y,(PUnidad+1)->Y,PUnidad->u,
       (PUnidad+1)->u,p+1,VCaux1);
       VEaux2[p]=ProdEscC(LandaSig,VCaux1);
    }

    SumaX(SumPJ,VEaux2,SumPJ);

    /* Landa actual*Parcial Xk+1 de Fk+1 */
    for(p=0; p<VE; p++)
    {
       ParcialX1qF((PUnidad-1)->Y,PUnidad->Y,(PUnidad-1)->u,
       PUnidad->u,p+1,VCaux2);
       VEaux1[p]=ProdEscC(PUnidad->landa,VCaux2);
```

```
   }
   /* Obtencion de la la parcial como suma de los terminos */
   SumaX(VEaux1,SumPJ,PUnidad->parJx);
}/* fin de la seccion de calculos complejos */
}/* fin de  CalcDelta*/


void ObtieneDelta(PRed)
TRed *PRed;
{
   Real VEaux[VE],VCaux[VC];
   TUnidad *UActual;
   SinSigno i,p;

   /* Calculo de parJx para etapa N */
    UActual=PRed->PtUni+PRed->N;  /* en la ultima cabecera */
    ParcialXnHn(UActual->Y,UActual->parJx);
   /*Calculo de landa N*/
    ParcialUdeH((UActual-1)->Y,UActual->u,UActual->landa);

   /* Landa actual*Parcial Xk+1 de Fk+1 */
   for(p=0; p<VE; p++)
   {
      ParcialX1qF((UActual-1)->Y,UActual->Y,(UActual-1)->u,
     UActual->u,p+1,VCaux);
      VEaux[p]=ProdEscC(UActual->landa,VCaux);
   }
   /* Obtencion de la la parcial como suma de los terminos */
   SumaX(VEaux,UActual->parJx,UActual->parJx);

   /*primero se calculan las deltas
      de todas las neuronas sin tocar los pesos*/
  for(i=PRed->N-1, UActual--;
 i>=1;
 i--, UActual--)
     CalcDelta(UActual,PRed->TraDese,i);
   /*Caso especial de la 1a. unidad*/
   CalcDelta(UActual,PRed->TraDese,0);
}
```

A.2.1.5 calgen.c

```
#include <string.h>
#include <math.h>
#define UPRECI 5.0e-17
#include "defred.h"

/*incluimos el ficheros de las funciones del problema para que se
   aproveche el nivel 3 optimizaciñn*/
#ifdef  Met1
#include PRO1
#else
#include PRO2
#endif

#include "funherra.h"

extern FILE *error;
extern volatile TLogico terminar;

/*********************************************************************
********* CALCULAMOS ***********************************************
*********************************************************************/

void CalculaUnidad(Unidad,xsal)
TUnidad *Unidad;
Real *xsal;   /* la devolvemos para que se almacene en siguiente */
{
   /* Dado xent y los pesos actuales se calcula
      el valor de salda xsal */

   SinSigno s,q,p,ns,ns_1,Lu=Unidad->L;
   Real *ypre;  /*Apuntara entrada del peso actual*/
   Real zcal,
    *wa=Unidad->Wa, /*puntero a los pesos*/
    *Yp=Unidad->Y,  /*Apuntara al a primera salida de la capa previa*/
    *ya=Yp+VE,   /*Apunta a salida de neurona que se esta calculando*/
    *za=Unidad->Z;  /*Apunta a Z de la neurona que se esta calulando*/

   /*El pto de entrada ya esta en Y*/

   /* bucle sobres las capas de la unidad */
   for( s=1;  /*empieza capa 1, termina en la penultima*/
        s<Lu; s++)
   {
      ns=Unidad->NeuC[s];
      ns_1=Unidad->NeuC[s-1];

      /* bucle sobre las neuronas de la capa*/
      for( q=1; q<=ns; q++)
      {
         zcal=(*wa); /* peso del BIAS */
         /* bucle sobre las entradas de la neurona */
         for(p=1, wa++, ypre=Yp;
             p<=ns_1;
     p++, wa++, ypre++)
                zcal+=(*wa)*(*ypre);

         *za=zcal; /* z de esta nuerona=suma entradas*pesos */
         /* salida=funcion de activacion sobre z para todos los
     tipos es tanh*/
         *ya=G(TH,zcal);
         ya++; /*pasamos a la siguiente neurona*/
         za++;
      }
      /*Tenemos que actualizar Yp al terminar con una capa
   debe apuntar a donde queda apuntando ypre*/
     Yp=ypre;
   }
   /*Para la ultima capa con salidaescalada. ns=VE*/
   ns_1=Unidad->NeuC[Lu-1];
   for( q=1; q<=VS; q++)
```

```
   {
      zcal=(*wa); /* peso del BIAS */
      for(p=1, wa++, ypre=Yp;
         p<=ns_1; p++, wa++, ypre++)
         zcal+=(*wa)*(*ypre);
      *za=zcal; /* z de esta nuerona=suma entradas*pesos */
      /*salida=funcion de activacion sobre z que es distinta segun
  cada neurona*/

      /*funcion activacion segun indicado en funsal*/
      *ya=G(Unidad->FunSal[q-1],zcal)*Unidad->RanSal[q-1];

      ya++; /*pasamos a la siguiente neurona*/
      za++;
   }
   /*el valor de la salida estara al final del array Y*/
      memcpy((void*)xsal,(void*)(Unidad->Y+(Unidad->TotNeu)+VE-VS),
      sizeof(Real)*VS);
      /* copia resultado a variable de salida */
}

CalRIni(PriUni,NuC)
TUnidad *PriUni;
SinSigno NuC;
{   /*Funcion que calcula red a partir de la unidad pasada
      y el numero de etapas indicadas dejendo resultado en la
      siguiente */

   TUnidad *UniAct;
   SinSigno k;
   for(k=1, UniAct=PriUni; k<=NuC; k++, UniAct++)
   {
#ifdef _Met1
      CalculaUnidad(UniAct,UniAct->u);
      F(UniAct->Y,UniAct->u,(UniAct+1)->Y);
#else
      CalculaUnidad(UniAct,(UniAct+1)->Y);
      F(UniAct->Y,(UniAct+1)->Y,UniAct->u,(UniAct+1)->u);
#endif
   }
}

#ifdef _Met1
   void CalculaRed(Red,Xo)
   TRed Red;
   Real *Xo;
   {
      /* Funcion que calcula todas las etapas a partir de pto Xo */
  memcpy((void*)(Red.PtUni->Y),(void*)Xo,sizeof(Real)*VE);
  CalRIni(Red.PtUni,Red.N);
   }
   Real CalCosto(red)
   TRed red;
   {
      /*Calcula el costo de la trayectoria en unidades de la red
    los comandos desde U1 a Un*/
      TUnidad *Ua;
      SinSigno k;
      Real Costo=0.0;
      for(k=0, Ua=red.PtUni; k<red.N; k++, Ua++)
      Costo+=H(Ua->Y,Ua->u,red.TraDese,k);
      Costo+=HN(Ua->Y);  /*Costo del punto final*/
      return Costo;
   }
#else
   void CalculaRed(Red,Xo,Uo)
   TRed Red;
   Real *Xo,*Uo;
   {
      /* Funcion que calcula todas las etapas a partir de pto Xo */
      /*Comd. inicial a la 1a. unidad Uo*/
      memcpy((void*)Red.PtUni->u,(void*)Uo,sizeof(Real)*VC);
      memcpy((void*)(Red.PtUni->Y),(void*)Xo,sizeof(Real)*VE);
      CalRIni(Red.PtUni,Red.N);
   }

   Real CalCosto(red)
   TRed red;
   {
      /*Calcula el costo de la trayectoria en unidades de la red
       los comandos desde U1 a Un*/
      TUnidad *Ua;
      SinSigno k;
      Real Costo=0.0;
      for(k=0, Ua=red.PtUni; k<red.N; k++, Ua++)
  Costo+=H(Ua->Y,(Ua+1)->u,red.TraDese,k);
      Costo+=HN(Ua->Y);  /*Costo del punto final*/
      return Costo;
   }
#endif

/*****************************************************************
*********  CORRECCION ***********************************************
*****************************************************************/
#ifdef _Met1
#   include "caldel1.h"
#else
#   include "caldel2.h"
#endif

void CalGrad(pRed,Sumar)  /*Valido para todos los metodos*/
TRed *pRed;
TLogico Sumar;
{
   /*Una vez tenemos los deltas calculamos el gradiente de todas las
      unidades y:
      - Si NO Sumar: lo devolvemos en Grad el anterior queda en Wp
      - Si Sumar: lo aAadimos al que esta en Grad
      - Se implemanta el escalado por capas (28-3-95)*/

   TUnidad *Uact;   /*Apunta unidad actual*/
   SinSigno k,s,q,p,ns,ns_1,pac;
   Real *pg,modulo=0.0,decEscala,escala=1;
   Real MaxCom=0.0;
   SinSigno umax,pmax;
```

```
    if(pRed->AlgDir & ESCALATOT)  /*debemos calcular total capas*/
    {
       Real TotCapas;
       for(k=0, Uact=pRed->PtUni, TotCapas=0.0; k<pRed->N; k++, Uact++)
    TotCapas+=Uact->L;
       decEscala=(pRed->PetMin-1)/TotCapas;
       escala=pRed->PetMin;
    }
    for(k=0, Uact=pRed->PtUni; k<pRed->N; k++, Uact++)
    {
       Real *PY=Uact->Y,      /*Apunta primera salida capa anterior*/
     *pd=Uact->Delta,   /*Apunta delta neurona actual*/
     *py;          /*Apuntara entrada a peso actual*/

       pg=Uact->Grad;       /*Apunta gradiente que se calcula*/
       /*Conservarmos gradiente anterior copiandolo*/
       if(! Sumar)
         memcpy(Uact->Wp,Uact->Grad,Uact->TotPes*sizeof(Real));
       for(s=1, pac=1; s<=Uact->L; s++)   /*Bucle sobre capas*/
       {
    if(pRed->AlgDir & ESCALARED)
       escala=(pRed->PetMin-1.0)*(Uact->L-s)/(Uact->L-1.0);
    else if (pRed->AlgDir & ESCALATOT)
       escala-=decEscala;
    ns=Uact->NeuC[s];
    ns_1=Uact->NeuC[s-1];
    for(q=1; q<=ns; q++) /*Bucle sobre neuronas de la capa*/
    {
       /*Para el Bias la entrada es 1*/
       *pg=(*pd)*escala+(Sumar?(*pg):0);
       modulo+=(*pg)*(*pg); /*cuadrado de la componente*/
       pg++; pac++;
       for(p=1, py=PY; p<=ns_1; p++, py++)
       {
          *pg=(*py)*(*pd)*escala+(Sumar?(*pg):0);
          modulo+=(*pg)*(*pg); /*cuadrado de la componente*/

          if((*pg)*(*pg)>MaxCom*MaxCom)
          {
            MaxCom=(*pg);
            umax=k;  /*ahora va a contener unidad del maximo*/
            pmax=pac; /*numero de neurona del maximo*/
          }
          pg++;
          pac++; /*Aumentamos el numero del peso*/
       }
       /*terminamos con la neurona->pasamos a la siguiente*/
       pd++;
    }
    /*Terminamos con la capa pasamos a la siguiente*/
    PY=py; /*py ya apunta primera salida capa siguiente*/
       }
    }
    modulo=sqrt(modulo);

    pRed->ModGra=modulo;
    pRed->MaxComG=MaxCom;
    pRed->Umax=umax;
    pRed->Pmax=pmax;

    return;
}


void CalDirBus(red,modA)
TRed *red;
Real modA;
{
   /*Aplica el algoritmo distintos para la direccion de busca:
   - Usar directamente el gradiente.
   - Conjugarlo Fletcher:
      * Sin reseteo
      * Con reseteo cada N
      * Con reseteo por proximidad a 0
   - Polak-Ribiere   */
   /*Tenemos en este momento:
   - modA el modulo del gradiente anterior
   - en Grad el gradiente actual
   - en ModGra el modulo del gradiente
   - En Wp el gradiente anterior
   - En DirB la direccion de busqueda previa  */
   /*Tenemos que obtener:
   - en DirB la nueva direccion de busqueda
   - en ModDirB el modulo de la direccion de busca  */

    SinSigno k,p;
    TUnidad *Ua;
    Real ModDirB,t0min,Num;
    Real *db,*pp,FacEta,*GradAnt,fa;
    if(!modA)
       FacEta=0.0;
    else switch (red->AlgDir & 0x0f)  /*Para quedarnos parte baja*/
    {
    case DIRGRA:
       FacEta=0.0;
       break;
    case DIRFLE:
       if ( (red->NumReset && !(red->NumItera%red->NumReset)) ||
     ((FacEta=((red->ModGra*red->ModGra)/(modA*modA)))<(1+red->Rango)
        && FacEta>(1-red->Rango) && !red->NumReset) )
    FacEta=0.0;
       break;
    case DIRPOL:
       Num=0.0;
       for(k=0, Ua=red->PtUni; k<red->N; k++, Ua++)
    for(p=1, pp=Ua->Grad, GradAnt=Ua->Wp;
       p<=Ua->TotPes; p++, pp++, GradAnt++)
          Num+=(*pp-*GradAnt)*(*pp);
       FacEta=Num/(modA*modA);
    }

    if(!FacEta)
    {
       for(k=0, Ua=red->PtUni; k<red->N; k++, Ua++)
       {
    for(p=1, pp=Ua->Grad, db=Ua->DirB;
```

```
      p<=Ua->TotPes; p++, pp++, db++)
         *db=-(*pp);
       }
       red->ModDB=ModDirB=red->ModGra;
    }
    else
    {
       ModDirB=0.0;
       for(k=0, Ua=red->PtUni; k<red->N; k++, Ua++)
       {
   for(p=1, pp=Ua->Grad, db=Ua->DirB;
      p<=Ua->TotPes; p++, pp++, db++)
   {
          *db=-(*pp)+FacEta*(*db);
          ModDirB+=(*db)*(*db);
   }
       }
       red->ModDB=ModDirB=sqrt(ModDirB);
    }
    /*Calculo del factor t0min y normalizacion de Dir. Busca*/
    for(k=0, Ua=red->PtUni, t0min=HUGE_VAL; k<red->N; k++, Ua++)
       for(p=1, pp=Ua->Wa, db=Ua->DirB;
   p<=Ua->TotPes; p++, pp++, db++)
       {
       *db/=ModDirB; /*Normalizamos el vector direccion*/
       if(*db<1e-250) continue;
       fa=(*pp/(*db));
       fa=(fa>0)?fa:-fa;
       if(fa<t0min)
          t0min=fa;
       }
    red->t0Min=t0min*UPRECI;
    red->Eta=FacEta;
}

void PuntPru(red,dez)
TRed red;
Real dez;
{
   /*Calcula pto siguiente en dirreccin de DirB
      con dezplazamiento dez*/
   TUnidad *Un;
   SinSigno k,p;
   Real *pp,*db,*pa;
   for(k=0, Un=red.PtUni; k<red.N; k++, Un++)
      for(p=1, pa=Un->Wa, pp=Un->Wp, db=Un->DirB;
   p<=Un->TotPes; p++, pa++, pp++, db++)
      *pa=(*pp)+dez*(*db);
}

#define MAXREF 5

void BuscaMin(red)
TRed *red;
{
/*definiciãn de la macro para calcular costo de un punto*/
#define COST(de)  (PuntPru(*red,(de)),CalRIni(red->PtUni,red->N),\
          CalCosto(*red))
/*  parametros delta de pto inicial y costo obtenido*/
   TUnidad *Uni;
   Real F1,F1p,F2,Fa,Fb,Fc,Fo;
   Real ta,tb,tc;
   Real *pa,t0,alfopt,Deno;
   SinSigno k,Ndiv=0,Nigua;

   /*Implementa distintos algoritmos de busqueda:
         - Paso Fijo
    - Busqueda sin peta => en cuanto Fc>Fb
    -     "     con  "   => hasta que Fc>%Fb+PetMin
   /*Distintas condiciones de comienzo:
    - Comenzar con paso fijo ALfa
    -    "       "   "   de la etapa anterior
    -    "       "  toMin  */
   /*Distintas condiciones de terminacion:
    - En cuanto se calcula Fo
    - En cuanto Fo<Fa  -> Reffiting
    - Hasta que Fo<=Fb -> Reffiting */

   /*Calculo Fa para el punto inicial.
      Como ya tenemos xk+1 y uk+1 basta aplicar H*/
   red->Fa=Fa=CalCosto(*red);
   ta=0.0;

   /*Salvamos pesos actuales copiando el contenido*/
   for(k=0, Uni=red->PtUni; k<red->N; k++, Uni++)
      memcpy(Uni->Wp,Uni->Wa,Uni->TotPes*sizeof(Real));


   if(red->AlgBus & BFIJA)
   {
      red->Fo=COST(red->Alfa);
      return;
   }
   if(red->AlgBus & BSimple)
   {
      if(red->AlfApli<1e-10)
   red->AlfApli=red->Alfa;
      red->Fo=COST(red->AlfApli);
      if(red->Fo<red->Fa) /*aumentamos el paso en c1*/
   red->AlfApli*=red->c1;
      else /*disminuimos el paso y reseteamos el conjugado*/
      {
   red->AlfApli*=red->c2;
   red->ModGra=0.0;
      }
      return;
   }

   /*Determinacion del paso inicial*/
   if((red->AlgBus & BIalp) && red->AlfApli)
      t0=red->AlfApli;
   else if(red->AlgBus & BIfija)
      t0=red->Alfa;
```

```
    else
       t0=red->t0Min;

    /*Comenzamos a buscar Fb y Fc*/
    F1=COST(t0);
    if(F1>Fa)
    {  /*Dividimos*/
       tc=t0;
       do
       {
    t0/=2; F2=COST(t0);
       } while(F2>=Fa && t0>=red->t0Min);
       if(F2>=Fa)  /*Error, no se encuentra Fb*/
       {
    /*Comenzamos desde t0min*/
    tb=t0=red->t0Min;
    Fb=COST(t0);
          fprintf(error,"\n%lu\t:No se encuentra Fb",red->NumItera);
   /* terminar=VERDAD;
    return;*/
       }
       else
       {
          Fb=F2; tb=t0; /*Tenemos Fb*/
          t0=tc/2;      /*Vemos si tc cumple las condiciones*/
       }
    }
    else
    {  /*ya tenemos Fb buscamos Fc*/
       tb=t0; Fb=F1;
    }
    Nigua=0;
    do
    {
       t0*=2; F1=COST(t0);
       if(F1<=Fb)
       {  /*Se detecta nuevo Fb*/
    tb=t0; Fb=F1;
       }
       else if (red->AlgBus & BsinPET)
    break; /*Fc si no queremos peta*/
       else if (red->AlgBus & Bsatura)
       {
    if(isnan(F1) || isinf(F1))
    {
       F1=F1p; t0/=2;
       break;
    }
    if(F1==F1p)
       Nigua++;
    else
       Nigua=0;
    if(Nigua==MAXIGUA)
       break;
    F1p=F1;
       }
       else if((F1-Fb)/Fb>red->PetMin)
    break; /*Si la peta es suficiente*/
    } while(1);
    Fc=F1; tc=t0;  /*El ultimo sera Fc*/

    Deno=(2*(Fa*(tb-tc)+Fb*(tc-ta)+Fc*(ta-tb)));
    if(Fa==Fb || !Deno) /*Fa=Fb=Fc => Tomamos Fb*/
    {          /*Aqui se deberia resetear el conjugado*/
       alfopt=tb; Fo=COST(alfopt);
       fprintf(error,"\n%lu\t:Se dio Fa=Fb=Fc,(Deno=0)",red->NumItera);
       terminar= terminar || (red->Eta==0);
       return;
    }
    alfopt=((Fa*(tb*tb-tc*tc)+Fb*(tc*tc-ta*ta)+Fc*(ta*ta-tb*tb))/Deno);

    Fo=COST(alfopt);
    if(red->AlgTer & FPRIMER)
    {
       red->Fo=Fo; red->AlfApli=alfopt; return;
    }

    /*Procedemos al reffiting*/
    Ndiv=0;
    while( (Fo>red->Fa) || ((red->AlgTer & FMFB)&&(Fo>Fb)) )
    {
       /*Nuevo alfopt formula 5-36*/
       /*Distinguir los casos*/
       if(alfopt>tb)
       {  /*Coger Fa Fb Fc=Fo*/
    tc=alfopt;
    Fc=Fo;
       }
       else
       {     /*Coger Fa=Fo Fb Fc*/
    ta=alfopt;
    Fa=Fo;
       }
       if(++Ndiv==MAXREF)
    break;
       Deno=(2*(Fa*(tb-tc)+Fb*(tc-ta)+Fc*(ta-tb)));
       if(Fa==Fb || !Deno)
       {  /*Fa==Fb==Fc => usamos Fb*/
    Ndiv=MAXREF;
       fprintf(error,"\n%lu\t:Se dio Fa=Fb=Fc,(Deno=0)",red->NumItera);
    break;
       }
       alfopt=((Fa*(tb*tb-tc*tc)+Fb*(tc*tc-ta*ta)+Fc*(ta*ta-tb*tb))/
         Deno);
       Fo=COST(alfopt);
    }
    if(Ndiv==MAXREF)
    {
       alfopt=tb;
       Fo=COST(alfopt);
    }
    red->Fo=Fo; red->AlfApli=alfopt;
    return;
}

#ifdef _Met1_
```

```
SinSigno PtosPro(Red,Ptos)
TRed Red;
Real *Ptos;
{
   Real *pt;
   RegionX Rex;
   SinSigno nptos,k,i,haymedio;
   /*Deberia explorar todas las esquinas de las regiones */
   Rex=Red.REntrena;

   /*en el caso general tentremos 2 elevado a VE ptos iniciales
     y el central*/

   nptos=0;
   haymedio=0;

   /*puntos extremos del hipercubo optimizando extremos iguales*/
   for(k=0, pt=Ptos; k<(1<<VE); k++)
   {
      nptos++;
      for(i=0; i<VE; i++, pt++)
   if(Rex.Xmax[i]==Rex.Xmin[i])
   {
      *pt=Rex.Xmax[i];
      k=(1<<i)|k; /*saltamos al caso maximo directamente*/
   }
   else
   {
     *pt=(k>>i)&1?Rex.Xmax[i]:Rex.Xmin[i];
     haymedio=1;
   }
   }

   /*punto central*/
   if(haymedio)
      for(i=0; i<VE; i++, pt++)
   *pt=(Rex.Xmax[i]+Rex.Xmin[i])/2;

   return nptos+haymedio;
}

void Promedia(PtRed,ptos,npt)
TRed *PtRed;
Real *ptos;
SinSigno npt;
{
   /*Deposita en Fa el promedio para los puntos extremos y central
      de la region de entrenamiento en x*/
   Real *pt,CosAcu;
   SinSigno i;

   for(i=1, pt=ptos, CosAcu=0; i<=npt; i++, pt+=VE)
   {
      CalculaRed(*PtRed,pt);
      CosAcu+=CalCosto(*PtRed);
   }
   PtRed->Fa=CosAcu/npt;
}
#else  /*para metodo 2 tambien comandos iniciales*/

SinSigno PtosPro(Red,Ptos)
TRed Red;
Real *Ptos;
{
   Real *pt=Ptos;
   RegionX Rex;
   RegionC Rec;
   SinSigno nptos,k,i,haymedio;

   /*Deberia explorar todas las esquinas de las regiones */
   Rex=Red.REntrena;
   Rec=Red.ComIni;

   /*en el caso general tentremos 2 elevado a VE ptos iniciales
     y el central*/

   nptos=0;
   haymedio=0;

   /*puntos extremos del hipercubo optimizando extremos iguales*/
   for(k=0, pt=Ptos; k<(1<<(VE+VC)); k++)
   {
      nptos++;
      for(i=0; i<VE; i++, pt++ )
   if(Rex.Xmax[i]==Rex.Xmin[i])
   {
      *pt=Rex.Xmax[i];
      k=(1<<i)|k; /*saltamos al caso maximo directamente*/
   }
   else
   {
     *pt=(k>>i)&1?Rex.Xmax[i]:Rex.Xmin[i];
      haymedio=1;
   }
      for(i=0; i<VC; i++, pt++)
   if(Rec.Cmax[i]==Rec.Cmin[i])
   {
      *pt=Rec.Cmax[i];
      k=(1<<(i+VE))|k; /*saltamos al caso maximo directamente*/
   }
   else
   {
     *pt=(k>>(i+VE))&1?Rec.Cmax[i]:Rec.Cmin[i];
      haymedio=1;
   }
   }


   /*punto central*/
   if (haymedio)
   {
      for(i=0; i<VE; i++, pt++)
   *pt=(Rex.Xmax[i]+Rex.Xmin[i])/2;
      for(i=0; i<VC; i++, pt++)
   *pt=(Rec.Cmax[i]+Rec.Cmin[i])/2;
```

```
    }
    return nptos+haymedio;

}

void Promedia(PtRed,ptos,npt)
TRed *PtRed;
Real *ptos;
SinSigno npt;
{
   /*Deposita en Fa el promedio para los puntos extremos y central
      de la region de entrenamiento en x*/
   Real *pt,CosAcu;
   SinSigno i;

   for(i=1, pt=ptos, CosAcu=0; i<=npt; i++, pt+=VE+VC)
   {
      CalculaRed(*PtRed,pt,pt+2);
      CosAcu+=CalCosto(*PtRed);
   }
   PtRed->Fa=CosAcu/npt;
}
#endif

void EligeXIni(Reg,pto)
RegionX Reg;
Real *pto;
{
   SinSigno i;
   Real *p;

   for(i=0,p=pto; i<VE; i++,p++)
   {
      if(Reg.Xmin[i]==Reg.Xmax[i])
         *p=Reg.Xmin[i];
      else
         *p=Reg.Xmin[i]+drand48()*(Reg.Xmax[i]-Reg.Xmin[i]);
   }
}

#ifndef  Met1
   void EligeComIni(Reg,Uo)
   RegionC Reg;
   Real *Uo;
   {
      SinSigno i;
      Real *p;

      for(i=0,p=Uo; i<VC; i++,p++)
      {
   if(Reg.Cmin[i]==Reg.Cmax[i])
      *p=Reg.Cmin[i];
   else
      *p=Reg.Cmin[i]+drand48()*(Reg.Cmax[i]-Reg.Cmin[i]);
      }
   }

   void UnEntre(PtRed)
   TRed *PtRed;
   {
      Real Xentre[VE],Uo[VC];
      unsigned long i;
      Real ModAnt;

      ModAnt=PtRed->ModGra;

      for(i=1; i<=PtRed->NumGrad; i++)
      {
   EligeXIni(PtRed->REntrena,Xentre);
   EligeComIni(PtRed->ComIni,Uo);
   CalculaRed(*PtRed,Xentre,Uo);
   ObtieneDelta(PtRed);
   CalGrad(PtRed,i!=1); /*Suma si es != de la primera iteracion*/
      }

      CalculaAlfa(PtRed);
      CalDirBus(PtRed,ModAnt);
    /*Se pasa a buscar el minimo en la direcion obtenida
 se pasa la red con lo que se podr- resetear gradiente conjugado*/
      BuscaMin(PtRed);  /*c3 ultimo alfa*/
   }
#else
   void UnEntre(PtRed)
   TRed *PtRed;
   {
      Real Xentre[VE];
      unsigned long i;
      Real ModAnt;

      ModAnt=PtRed->ModGra;

      for(i=1; i<=PtRed->NumGrad; i++)
      {
   EligeXIni(PtRed->REntrena,Xentre);
   CalculaRed(*PtRed,Xentre);
   ObtieneDelta(PtRed);
   CalGrad(PtRed,i!=1); /*Suma si es != de la primera iteracion*/
      }

      if(!PtRed->ModGra)
   fprintf(error,"\nModulo del Gradiente 0.0.(it:%lu)"
                           ,PtRed->NumItera);
      else
      {
   CalculaAlfa(PtRed);
   CalDirBus(PtRed,ModAnt);
       /*Se pasa a buscar el minimo en la direcion obtenida
     se pasa la red con lo que se podr- resetear gradiente conjugado*/
   BuscaMin(PtRed);  /*c3 ultimo alfa*/
      }
   }
#endif
```

A.2.1.6 entregen.c

```
#include <stdio.h>
#include <string.h>
#include "defred.h"

extern volatile TLogico terminar;
SinSigno PtosPro();
unsigned EntraIni();
void UnEntre(),Promedia(),SalEstMat();


void Entrena(PtRed)
TRed *PtRed;
{
   unsigned long NumEnt;

   printf("\nNumero de Entranamientos realizados hasta ahora: %lu",
         PtRed->NumItera);

   printf("\nHasta donde quieres realizar ahora?");
   scanf("%lu",&NumEnt);


   printf("\nSe van arealizar %lu ciclos.\n",NumEnt-PtRed->NumItera);

   terminar=FALSO; /*esta variable la puede cambiar seÀal USR1*/
   while(!terminar && PtRed->NumItera<NumEnt)
   {
      PtRed->NumItera++;
      /* Bucle de entrenamientos */
      UnEntre(PtRed);
   }
   printf("\nTerminada serie de entrenamientos\n");
}


#define CreaEspDatos(pr,pd) if(!(pd=malloc((size_t)*(pr->PtUni->Wa-1))))
return;
#define CopiaDatos(pr,pd) memcpy(pd,pr->PtUni->Wa,(size_t)*(pr->PtUni->Wa-
1))
#define DatosCopia(pr,pd) memcpy(pr->PtUni->Wa,pd,(size_t)*(pr->PtUni->Wa-
1))
EntreSerie(PtRed)
TRed *PtRed;
{
   TLogico SalPes,SalGra,SalY,SalZ,SalD;
   unsigned long MaxItera,Intervalo,Factor;
   unsigned long MusDat,ndat;

   Real xi[MAXPTINI*(VE+VC)];
   char c,Pref[6],NomV[20],Nomf[20],NomMej[20];
   TTrayec *tr[MAXPTINI];

   SinSigno NumIni,n;
   FILE *FiMat;  /*FiEst ya nu se usa => se salva todo en FiMat*/
   Real FproMin,*datos,*dta,*PtDt,*PtTmp;
#ifdef  Met1
   Real PtPr[((1<<VE)+1)*VE];
#else
   Real PtPr[((1<<(VC+VE))+1)*(VE+VC)];
#endif
   SinSigno NptPr;
   TRed Rmin;

   NptPr=PtosPro(*PtRed,PtPr);
   MuesPtos(PtPr,NptPr);
   Promedia(PtRed,PtPr,NptPr);
   FproMin=PtRed->Fa;  /*Inicialmente el costo promedio actual*/
   CreaEspDatos(PtRed,PtDt);
   CopiaDatos(PtRed,PtDt);
   Rmin=*PtRed;


   /*Obtencion de las condiciones de entrenamiento*/
   printf("\nRealizacion de series de entrenamientos");
   printf("\n con salidas parciales a ficheros");

   printf("\nNumero entrenos hasta donde se desa llegar:\n");
   scanf("%lu",&MaxItera);

   printf("Cada cuantos se realiza testeo:");
   scanf("%lu",&Intervalo);

   printf("\n Prefijo ficheros (max 5 caracteres):");
   scanf("%s",Pref);

   printf("Factor de division para el sufijo:");
   scanf("%lu",&Factor);

   printf("\n Periodo de muestreo de los datos:");
   scanf("%lu",&MusDat);

   printf("\n Deseas salvar los pesos (s/[n])");
   while((c=(getchar() | 0x20))!='s'  && c!='n');
   SalPes=(c=='s');
   printf("\n Deseas salvar el gradiente  (s/[n])");
   while((c=(getchar() | 0x20))!='s'  && c!='n');
   SalGra=(c=='s');
   printf("\n Deseas salvar los Y (s/[n])");
   while((c=(getchar() | 0x20))!='s'  && c!='n');
   SalY=(c=='s');
   printf("\n Deseas salvar los Z (s/[n])");
   while((c=(getchar() | 0x20))!='s'  && c!='n');
   SalZ=(c=='s');
   printf("\n Deseas salvar los Dalta (s/[n])");
   while((c=(getchar() | 0x20))!='s'  && c!='n');
   SalD=(c=='s');

   printf("\n Puntos sobre los que realizar testeo:");
   NumIni=EntraIni(*PtRed,xi);

   printf("\nParametros entrados:");
```

```
printf("\nNumero maximo entrenos=%lu",MaxItera);
printf("\nIntervalo de muestra=%lu",Intervalo);
printf("\nPrefijo de los ficheros: '%s'",Pref);
printf("\nFactor para sufijo= %lu",Factor);
printf("\nPeriodo de muestreo de datos= %lu",MusDat);
printf("\n%s se salvaran los Pesos",SalPes?"SI":"NO");
printf("\n%s se salvaran los Gradientes",SalGra?"SI":"NO");
printf("\n%s se salvaran los Y",SalY?"SI":"NO");
printf("\n%s se salvaran los Z",SalZ?"SI":"NO");
printf("\n%s se salvaran los Delta",SalD?"SI":"NO");

printf("\nDeseas continuar?(s/n)");
while( (c=getchar()|0x20)!='s' && c!='n') ;
if (c=='n')
   return;

/*Intentamos abrir el fichero de matlab aqui iran todos los datos*/
sprintf(NomV,"%s%lu",Pref,MaxItera/Factor);
strcpy(Nomf,NomV);  strcat(Nomf,".mat");
if(!(FiMat=fopen(Nomf,"w")))
{
   printf("\nNo puedo abrir fichero de resultados:\n%s",Nomf);
   return;
}

/*Intentamos abrir el fichero de matlab estado es el mismo que para
   el resto*/

for(n=0; n<NumIni ;n++)
   if(!(tr[n]=malloc((PtRed->N+1)*sizeof(TTrayec))))
   {
      printf("\nNO Obtenida memoria para las trayectorias");
      return;
   }

/*Memoria para los datos */
if(!(datos=malloc((size_t)(Intervalo/MusDat*sizeof(Real)*NDAT+10))))
{
   printf("\nNo Obtenida memoria para los datos");
   return;
}

/* Ciclo de entrenamientos */
terminar=FALSO; /*esta variable la puede cambiar seÀal USR1*/
while(!terminar && PtRed->NumItera<MaxItera)
{
   dta=datos;
   ndat=0;
   do
   {
      PtRed->NumItera++;
      /* Bucle de entrenamientos */
UnEntre(PtRed);

Promedia(PtRed,PtPr,NptPr);

      /*comprobamos si se trata del mejor costo delmomento*/
if((PtRed->Fa)<FproMin)
{
   CopiaDatos(PtRed,PtDt);
   Rmin=*PtRed;
   FproMin=PtRed->Fa;
}

if(! (PtRed->NumItera%MusDat) )
{
   /*Copiar los datos*/
   *(dta++)=(Real)PtRed->NumItera;
   *(dta++)=PtRed->ModGra;
   *(dta++)=PtRed->Eta;
   *(dta++)=PtRed->ModDB;
   *(dta++)=PtRed->AlfApli;
   *(dta++)=PtRed->Fo;
   *(dta++)=PtRed->Fa;
   *(dta++)=PtRed->t0Min;
   *(dta++)=PtRed->MaxComG;
   *(dta++)=(Real)PtRed->Umax;
   *(dta++)=(Real)PtRed->Pmax;
   ndat++;

} /*Fin del if divisible*/

   } while(!terminar && (PtRed->NumItera % Intervalo));
     /* Mientras no divisible */

   /* debemos calcular para los iniciales y salvar */
   CompTra(*PtRed,xi,tr,NumIni);
   sprintf(NomV,"%s%lu",Pref,PtRed->NumItera/Factor);
   if (!SalTraMat(*PtRed,tr,NumIni,FiMat,NomV,datos,ndat))
   {
      printf("\n Problemas al salvar fichero %s ",Nomf);
      return ;
   }
   /*El estado se salva en el momento de la trayectoria*/
   SalEstMat(*PtRed,FiMat,NomV,SalPes,SalGra,SalY,SalZ,SalD);

} /*del While de entrenamienteo*/

/*Preparamos nombres de los mejores*/
strcpy(NomMej,Pref);    strcat(NomMej,"MEJ.rb");
PtTmp=PtRed->PtUni->Wa;
PtRed->PtUni->Wa=PtDt;
SalvaRed(Rmin,NomMej);
PtRed->PtUni->Wa=PtTmp;
CreaEspDatos(PtRed,PtTmp);
CopiaDatos(PtRed,PtTmp);
DatosCopia(PtRed,PtDt);
/*Calculamos mejor trayectoria y lo salvamos ya en FiMat*/
CompTra(Rmin,xi,tr,NumIni);
strcpy(NomMej,Pref);    strcat(NomMej,"MEJ");
if (!SalTraMat(Rmin,tr,NumIni,FiMat,NomMej,(Real*)0,0))
{
   printf("\n Problemas al salvar fichero %s ",Nomf);
   return ;
}
/*El estado se salva en el momento de la trayectoria*/
```

```
   SalEstMat(Rmin,FiMat,NomMej,SalPes,SalGra,SalY,SalZ,SalD);
   DatosCopia(PtRed,PtTmp);
   free(PtDt); free(PtTmp);


   for(n=0; n<NumIni ;n++)
      free((void *)tr[n]);
   free((void*)datos);
   fclose(FiMat);
   return;
}
```

A.2.1.7 fiogen.c

```
#include <stdio.h>
#include <string.h>
#include "defred.h"

int Salva(Red,NomFS)
TRed Red;
char *NomFS;
{
   char c;

   printf("\n Salvar la red en estado actual.");
   if (*NomFS)
   {
      printf("\nQuieres salvar en fichero '%s' ?(s/n)",NomFS);

      while( (c=getchar()|0x20)!='s' &&  c!='n') ;
      if (c=='s')
         return SalvaRed(Red,NomFS);
   }

   printf("\nNombre del fichero:");
   scanf("%s",NomFS);

   while(*NomFS && ! SalvaRed(Red,NomFS))
   {
      printf("No puedo abrir ese fichero.Nombre:");
      scanf("%s",NomFS);
   }
   return 1;
}


int SalvaRed(Red,NomF)
TRed Red;
char *NomF;
{   /*La modificacion 14/12/94 hace que los datos se salven
   recorriendo el bloque de memoria*/
    /*En esta version existe la posibilidad de que exista trayectoria
    deseada, Se salvara tras la cabezera. Version >0xA001*/
   FILE *fd;
   TUnidad *UAct;
   SinSigno i;
   size_t TamTra;
   long Vnum=VERNUM;
   Real *PtDt;

   if(! (fd=fopen(NomF,"w")))
      return 0;  /* no pudimos abrir fichero */

   /*Primero el numero de version*/
   fwrite((void*)&Vnum,sizeof(long),1,fd);

   /*Despues Red con todos sus parametros*/
   fwrite((void *)&Red,sizeof(TRed),1,fd);

   /*ahora el vectror trayectoria si existe*/
   /* TamaÃo del vector almacenado el direccion anterior*/
   if(Red.TraDese)
   {
      TamTra=(size_t)*(Red.TraDese-1);
      fwrite(&TamTra,sizeof(size_t),1,fd);
      fwrite((void*)Red.TraDese,sizeof(Real),TamTra,fd);
   }
   else
   {
      TamTra=0;
      fwrite(&TamTra,sizeof(size_t),1,fd);
   }
   /* para cada etapa (NO N):
   - el numero de capas
   - el no.total de neuronas y capas
   - el rango de salida
   - la funcion de salida*/
   for(i=0, UAct=Red.PtUni; i<Red.N; i++,UAct++)
   {
      fwrite((void *)&UAct->L,sizeof(SinSigno),1,fd);
      fwrite((void *)&UAct->TotNeu,sizeof(SinSigno),1,fd);
      fwrite((void *)&UAct->TotPes,sizeof(SinSigno),1,fd);
      fwrite((void *)UAct->RanSal,sizeof(Real),VS,fd);
      fwrite((void *)UAct->FunSal,sizeof(SinSigno),VS,fd);
   }


   /* para cada unidad: (NO la N)
         - neuronas por capa
         - Pesos actuales
         - Gradiente
         - Direccion de busqueda  */
   for(i=0, UAct=Red.PtUni, PtDt=UAct->Wa; i<Red.N;
   PtDt+=UAct->TotPes*3, i++, UAct++)
   {
     /*Salvar numero de neuronas incluso nivel 0 */
      fwrite((void *)UAct->NeuC,sizeof(SinSigno),(size_t)UAct->L+1,fd);
     /*Salvar Wa, Grad y DirB todo de golpe*/
      fwrite((void *)PtDt,sizeof(Real),(size_t)UAct->TotPes*3,fd);
   }

   fclose(fd);
```

```
   return 1; /* terminacion correcta */
}

int CargaRed(PtRed,NomF)
TRed *PtRed;
char *NomF;
{
   /*Modificado 14/12/93 para cargar todos los Wa,grad,DirBus en unico
      bloque contiguo*/
   FILE *fe;
   TUnidad *UAct;
   long Vnum;
   SinSigno i,k,PesosTotal;
   Real *PtDt;
   size_t TamTra;

   if(! (fe=fopen(NomF,"r")))
      return 0;  /* no pudimos abrir fichero */

   /*Primero leemos el numero de version*/
   fread((void*)&Vnum,sizeof(long),1,fe);
   /*Comprobamos que corresponde*/
   if(Vnum != VERNUM) /*version sin trayectoria deseada*/
   {
      printf("\n Numero de Version no soportada");
      exit(1);
   }

   /*Cargamos estructura red*/
   fread(PtRed,sizeof(TRed),1,fe);

   /*Cargamos trayectoria si existe*/
   fread(&TamTra,sizeof(size_t),1,fe);
   if(TamTra)
      if(!(PtRed->TraDese=malloc(sizeof(Real)*(TamTra+1))) )
   return 0;
      else
      {
   *(PtRed->TraDese)=TamTra;
   PtRed->TraDese++;
   fread((void*)PtRed->TraDese,sizeof(Real),TamTra,fe);
      }

   /* Pedimos memoria para el array de unidades */
   if(! (PtRed->PtUni=malloc((PtRed->N+1)*sizeof(TUnidad))))
      return 0;  /* no la obtubimos */

   /* Leemos numero de cada etapa:
   - el numero de capas
   - el no. total de neuronas y pesos
   - el rango de salida   */
   for(i=0, UAct=PtRed->PtUni; i<PtRed->N; i++,UAct++)
   {
      fread((void *)&UAct->L,sizeof(SinSigno),1,fe);
      fread((void *)&UAct->TotNeu,sizeof(SinSigno),1,fe);
      fread((void *)&UAct->TotPes,sizeof(SinSigno),1,fe);
      fread((void *)UAct->RanSal,sizeof(Real),VS,fe);
      if(Vnum<0xA001) /*version sin trayectoria deseada*/
   for(k=0; k<VS; k++)
      UAct->FunSal[k]=TH;  /*si no especficado tanh*/
      else
   fread((void*)UAct->FunSal,sizeof(SinSigno),VS,fe);
   }

   UAct->L=0; UAct->TotNeu=0; UAct->TotPes=0;


   /*Calculamos el total de pesos del sistema*/
   for(i=0, PesosTotal=0, UAct=PtRed->PtUni;
      i<PtRed->N; PesosTotal+=UAct->TotPes,i++,UAct++);
   /*Pedimos memoria para la estructura Wa+Grad+DirBus*/
   PtDt=malloc((PesosTotal*3+1)*sizeof(Real));
   /*en la primera posiciãn guardamos el tamaÃo de datos*/
   *PtDt=(Real)((PesosTotal*3)*sizeof(Real));

   /* para cada unidad:  (No la N-sima)
               neuronas por capa
               pesos actuales
               pesos previos   */
   for(i=0, UAct=PtRed->PtUni,PtDt++; i<PtRed->N; i++,UAct++)
   {

     /*Pedimos memoria para array NeuC*/
      if(! (UAct->NeuC=malloc((UAct->L+1)*sizeof(SinSigno))))
         return 0; /* no tenemos memoria */
     /*Leemos numero de neuronas incluso nivel 0 */
      fread((void *)UAct->NeuC,sizeof(SinSigno),(size_t)UAct->L+1,fe);

      /*Conseguimos memoria para los arrays*/
   /*Para Y=no. neuronas+ VE*/
      if(!(UAct->Y=malloc((UAct->TotNeu+VE)*sizeof(Real))) ||
   /*Para Z y Delta = no. de neuronas*/
   !(UAct->Z=malloc(UAct->TotNeu*sizeof(Real))) ||
   !(UAct->Delta=malloc(UAct->TotNeu*sizeof(Real))) ||
   /*Para los pesos y direccionB = no. de pesos*/
   /*Wa grad DirB: ya obtenidas*/
   !(UAct->Wp=malloc(UAct->TotPes*sizeof(Real))) )
      return 0; /*No obtuvimos memoria*/
      /*Asignamos los valores de los punteros sobre el bloque*/
      UAct->Wa=PtDt;
      PtDt+=UAct->TotPes;
      UAct->Grad=PtDt; PtDt+=UAct->TotPes;
      UAct->DirB=PtDt; PtDt+=UAct->TotPes;

#ifdef  Met1
      /*Conectamos u al final de Y*/
      UAct->u=UAct->Y+(UAct->TotNeu+VE-VS);
#endif

      /*Leemos unicamente los arrays:
                  -de pesos actuales
            -de gradiente
            -de direccion de busqueda*/
         fread((void *)UAct->Wa,sizeof(Real),(size_t)UAct->TotPes,fe);
         fread((void *)UAct->Grad,sizeof(Real),(size_t)UAct->TotPes,fe);
```

```
            fread((void *)UAct->DirB,sizeof(Real),(size_t)UAct->TotPes,fe);
      }
      /*Para la ultima capa*/
         /*Memoria para array Y, solo para que contenga estado*/
         if( !(UAct->Y=malloc(VE*sizeof(Real))) )
      return 0;

      /*no se comprueba que la hayan leido los datos correctamente*/
      fclose(fe);
      return 1; /* terminacion correcta */
}
```

A.2.1.8 iogen.c

```
#include "defred.h"
#include <stdio.h>
#include <string.h>


void Libera(PtRed)
TRed *PtRed;
{
   /* Procedimiento para liberar toda la memoria */
   TUnidad *Uact;
   SinSigno i;

   free((void*)(PtRed->TraDese-1)); /*usamos estructura similar a Wa*/
   /*Por modificacion 14/12/93 todos los Wa,grad y DirB juntos*/
   free((void *)(PtRed->PtUni->Wa-1));
   /*Wa apunta al 2o. elemento del bloque. El 1o. mantiene su tamaÀo*/

   for(i=0, Uact=PtRed->PtUni;
      i<PtRed->N; /* ultima uni solo tiene Y */
      i++, Uact++)
      {
   free((void*)Uact->NeuC);
         free((void*)Uact->Y);
   free((void*)Uact->Z);
   free((void*)Uact->Delta);
   free((void*)Uact->Wp);
      }
   free((void *)Uact->Y); /*La ultima solo tiene memoria en Y*/
   /* liberamos todas las cabeceras de unidad */
   free((void *)PtRed->PtUni);
}

void PresentaUnidades(Red)
TRed Red;
{
   /* Funcion que presenta,en columna, las unidades
      con su numero de capas */
   TUnidad *PtU;
   SinSigno i;

   printf("\nUnid:  capas:");

   for(i=0, PtU=Red.PtUni; i<=Red.N; i++, PtU++)
      printf("\n %4hu  %4hu",i,PtU->L);
}


PresentaCapas(Red)
TRed Red;
{
   /* Funcion que presenta,por filas,a las unidades
      con su numero de neurona por capa */
   TUnidad *PtU;
   SinSigno i,j;


   printf("\nNumero de neuronas por unidad y capa");
   printf("\nUnidad:");

   for(i=0, PtU=Red.PtUni; i<=Red.N; i++, PtU++)
   {
      printf("\n %4hu      ",i);
      for(j=1; j<=PtU->L; j++)
         printf(" %4hu ",PtU->NeuC[j]);
   }
}

void EntraRango(VecR)
Real VecR[VS];
{
   SinSigno i;
   Real ra;
   for(i=0;i<VS;i++)
   {
      do {
      printf("\nX[%hu]=",i+1);
      } while(!scanf("%lf",&ra));
      VecR[i]=ra;
   }
}

void EntraFuncion(VecF)
SinSigno VecF[VE];
{ /*entra funciones: TH,SG,LI*/
   SinSigno i;
   char c;

   for(i=0;i<VS;i++)
   {
      printf("\nElige Tanh, Sigmoide, Lineal: X[%hu]=",i+1);
      while((c=(getchar()|0x20))!='t' && c!='s' && c!='l');
      switch(c)
      {
      case 't': VecF[i]=TH; break;
      case 's': VecF[i]=SG; break;
      case 'l': VecF[i]=LI; break;
      }
   }
}
```

```
Real cm0[VC];  /* valor inicial a 0 */
int CreaEstructura(PtRed)
TRed *PtRed;
{
   SinSigno LTip,IUni,ICapa,NeuTip,i,j,ent,PesosTotal;
   TUnidad *PtU;
   Real *PtDt;

   do
   {
      printf("\nNumero de etapas(>0): ");
      scanf("%hu",&PtRed->N);
   } while(!PtRed->N);
   /* tendremos nodo 0 hasta N = N+1*/
   if (!(PtRed->PtUni=malloc((PtRed->N+1)*sizeof(TUnidad))))
      return 0; /* no se consiguio memoria */

   do
   {
      printf("\nNumero tipico de capas por unidad(>0): ");
      scanf("%hu",&LTip);
   } while(!LTip);

   for(i=0, PtU=PtRed->PtUni; i<PtRed->N; i++, PtU++)
      PtU->L=LTip;
   PtU->L = (SinSigno)0;

   do
   {
      PresentaUnidades(*PtRed);
      printf("\nEntra numero de unidad a modificar(>N para terminar):");
      scanf("%hu",&IUni);
      if (IUni<=PtRed->N)
      {
         do
         {
            printf("\nNumero de capas para la unidad %hu= ",IUni);
            scanf("%hu",&ent);
         } while(!ent);
         (PtRed->PtUni+IUni-1)->L=ent;
      }
   } while(IUni<=PtRed->N);

   /*Obtenemos memoria para el array de no. de neuronas*/
   for(i=0, PtU=PtRed->PtUni; i<PtRed->N; i++, PtU++)
      if(!(PtU->NeuC=malloc((PtU->L+1)*sizeof(SinSigno))))
         return 0;

   do
   {
   printf("\n\nNumero tipico de Neuronas por capa(>0)= ");
   scanf("%hu",&NeuTip);
   } while(!NeuTip);

   for(i=0, PtU=PtRed->PtUni; i<PtRed->N; i++, PtU++)
   {
      PtU->NeuC[0]=VE;  /* Fijadas = al no. entradas  */
      for( j=1; j<PtU->L; j++)
         PtU->NeuC[j]=NeuTip;
      PtU->NeuC[PtU->L]=VS;/*Neuronas ultima capa=Variables de estado*/
   }

   do
   {
      PresentaCapas(*PtRed);
      printf("\nUnidad y capa a modificar(>N para terminar):");
      printf("\nUnidad="); scanf("%hu",&IUni);
      printf("Capa=");   scanf("%hu",&ICapa);
      if((IUni<PtRed->N) &&
         ICapa && (ICapa<(PtRed->PtUni+IUni)->L))
      {
         do
         {
           printf("Numero deseado de neuronas (U:%hu,C:%hu)(>0)=",
           IUni,ICapa);
           scanf("%hu",&ent);
         } while(!ent);
         (PtRed->PtUni+IUni)->NeuC[ICapa]=ent;
      }
   } while(IUni<=PtRed->N );


   /* Obtenemos memoria para los arrays de cada unidad*/
   for(i=0, PtU=PtRed->PtUni, PesosTotal=0; i<PtRed->N;
      PesosTotal+=PtU->TotPes, i++, PtU++)
   {
      PtU->TotNeu=0;
      PtU->TotPes=0;
      for(j=1;j<=PtU->L;j++)
      {
   PtU->TotNeu+=PtU->NeuC[j];
      /*En cada capa: pesos=Numero_Neuronas*Pesos_en_cada*/
   PtU->TotPes+=PtU->NeuC[j]*(PtU->NeuC[j-1]+1);
      }
      /*Para Y->Numero de neuronas + entradas*/
      if(!( PtU->Y=malloc((PtU->TotNeu+VE)*sizeof(Real)) ))
            return 0; /* No memoria para pesos */
      /*Para Z-> uno por neurona*/
      if(!( PtU->Z=malloc(PtU->TotNeu*sizeof(Real)) ))
            return 0; /* No memoria para pesos */
      /*Para delta-> una por neurona*/
      if(!( PtU->Delta=malloc(PtU->TotNeu*sizeof(Real)) ))
            return 0; /* No memoria para pesos */
      /*para Wp  una por peso*/
      if( !(PtU->Wp=malloc(PtU->TotPes*sizeof(Real))) )
            return 0; /* No memoria para pesos */
      /*para Wa Grad y DirB se piden despues en unico bloque*/

#ifdef  Met1
      PtU->u=PtU->Y+(PtU->TotNeu+VE-VC);
#endif
   }
   /*Bolque comun*/
   if( !(PtDt=malloc((PesosTotal*3+1)*sizeof(Real))) )
```

```
      return 0;
   *PtDt=(Real)((PesosTotal*3)*sizeof(Real)); /*tamaÀo 1a. posicion*/
   /*Asignacion de los punteros*/
   for(i=0, PtU=PtRed->PtUni, PtDt++; i<PtRed->N; i++, PtU++)
   {
       PtU->Wa=PtDt; PtDt+=PtU->TotPes;
       PtU->Grad=PtDt; PtDt+=PtU->TotPes;
       PtU->DirB=PtDt; PtDt+=PtU->TotPes;
   }

   /*para la ultima unidad (ficticia)*/
      PtU->TotNeu=0;
      PtU->TotPes=0;
      /*Para Y->solo entradas*/
      if(!( PtU->Y=malloc(VE*sizeof(Real)) ))
     return 0; /* No memoria para pesos */

      /*landa sera siempre cero*/
      memcpy((void *)PtU->landa,(void *)cm0,sizeof(Real)*VC);

   return 1;
}


void IniciaPesos(Red)
TRed Red;
{
   /* Procedimiento para poner valor inicial a los pesos */
   TUnidad *Uact;

   SinSigno n,l;
   Real *Pa,*Pp,cini,vini,vfin,dif;
   char c;
   TLogico Iguales;

   /* Se permite inicializar pesos a valor constante o aleatorio */

   printf("\nColocacion de Valores iniciales de los pesos");
   printf("\n   Valor igual o aleatorio?(i/a)");
   while((c=getchar())!='i' && c!='a');
   if (c=='i')
   {
      printf("\nValor constante a colocar?=");
      scanf("%lg",&cini);
      Iguales=VERDAD;
   }
   else
   {
      printf("\nValor minimo posible?=");
      scanf("%lg",&vini);
      printf("\nValor maximo posible?=");
      scanf("%lg",&vfin);
      dif=vfin-vini;
      Iguales = FALSO;
   }
   printf("\n*** Colocando los pesos ***");
   for(n=0, Uact=Red.PtUni;
       n<Red.N;  /* ultima unidad sin red */
       n++, Uact++)
      for( l=1, Pa=Uact->Wa, Pp=Uact->Wp;
          l<=Uact->TotPes;
          l++, Pa++, Pp++)
      {
  *Pa=Iguales ? cini
     :(drand48()*dif+vini);
  (*Pp)=(*Pa);
      }
}

void CargaPesM(Ptred)
TRed *Ptred;
{
   /*Carga los pesos en la red desde matriz en fichero matlab*/
   FILE *fm;
   CabezaMat Cabeza;
   char c,NomF[30],NomV[20];
   int sfk,sfr;
   SinSigno n;
   TUnidad *uni;
   do
   {
      printf("\nNombre completo del fichero:");
      scanf("%s",NomF);
   }
   while( *NomF && !(fm=fopen(NomF,"r")));

   if(!*NomF)
   {
      printf("\nNo se ha especificado nombre =>");
      printf("No se han cargado pesos");
      return;
   }

   do
   {
      fread((void*)&Cabeza,sizeof(CabezaMat),1,fm);
      fread((void*)NomV,sizeof(char),(size_t)Cabeza.TamNom,fm);

      printf("\n Cargar variable:<%s>",NomV);
      while((c=getchar())!='s' && c!='n');
      if(c=='n')
         /*Nos saltamos los datos*/
    sfk=fseek(fm,(long int)(Cabeza.NumFilas*Cabeza.NumColum*
        sizeof(Real)),SEEK_CUR);
   }
   while(c=='n' && !sfk);

   if(c!='s')
   {
      printf("\nNo mas variables. No se cargan pesos");
      return;
   }

   if(Cabeza.NumColum!=Ptred->N)
   {
```

```
      printf("\nEl numero de columnas no coinciden con el nÃmero de ");
      printf("unidades.\n No se cargan pesos");
      return;
   }

   uni=Ptred->PtUni;
   if(Cabeza.NumFilas!=uni->TotPes)
   {
      printf("\nNumero de filas no coincide con el numero de pesos");
      printf("\nNo se cargan los pesos");
      return;
   }

   /*Cargamos los pesos comprobando que todas las redes son iguales*/
   for(n=1, sfr=1; n<=Ptred->N && sfr; n++, uni++)
   {
      if(Cabeza.NumFilas!=uni->TotPes)
      {
  printf("\nNumero de filas no coincide con el numero de pesos");
  printf("\nNo se cargan los pesos");
  return;
      }
      sfr=fread((void*)uni->Wa,sizeof(Real),(size_t)uni->TotPes,fm);
      if(!sfr)
      {
 printf("\nHubo problemas cargando los pesos. Semimodificados.");
  return;
      }
   }
   fclose(fm);

}


void CargaTraDesMat(Ptred)
TRed *Ptred;
{
   /*Carga la trayectoria deseada desde matriz en fichero matlab*/
   FILE *fm;
   CabezaMat Cabeza;
   char c,NomF[30],NomV[20];
   int sfk;
   Real *t,*ta;
   SinSigno v,n;
   size_t TamTra;

   /*En principo eliminamos posible trajectoria*/
   if(Ptred->TraDese /*!=(Real*)void*/)
      free((void *)(Ptred->TraDese-1));
   do
   {
      printf("\nNombre completo del fichero en Matlab:");
      scanf("%s",NomF);
   }
   while( *NomF && !(fm=fopen(NomF,"r")));

   if(!*NomF)
   {
      printf("\nNo se ha especificado nombre =>");
      printf("No existe trayectoria deseada");
      return;
   }

   do
   {
      fread((void*)&Cabeza,sizeof(CabezaMat),1,fm);
      fread((void*)NomV,sizeof(char),(size_t)Cabeza.TamNom,fm);

      printf("\n Cargar variable:<%s>",NomV);
      while((c=getchar())!='s' && c!='n');
      if(c=='n')
         /*Nos saltamos los datos*/
  sfk=fseek(fm,(long int)(Cabeza.NumFilas*Cabeza.NumColum*
        sizeof(Real)),SEEK_CUR);
   }
   while(c=='n' && !sfk);

   if(c!='s')
   {
      printf("\nNo mas variables. No se pesa trayectoria");
      return;
   }

   if(Cabeza.NumColum!=VE)
   {
      printf("\nEl numero de columnas no coinciden con el nÃmero de ");
      printf("Variables de entrada.\n No se cargan Trayectoria");
      return;
   }
/*ahora el numero de filas no esta controlado
para permitir zona prohibida*/

   /*Pedimos memoria para trayectoria*/
   TamTra=VE*(Cabeza.NumFilas);
   if(!(Ptred->TraDese=malloc(sizeof(Real)*(TamTra+1))) )
   {
      printf("\nNo se pudo obtener memoria de trayectoria");
      return;
   }
   *Ptred->TraDese=TamTra;
   Ptred->TraDese++;
   if((Cabeza.tipo%1000)/100)
      /*orientado por filas=> cargamos directamente*/
      fread((void*)Ptred->TraDese,sizeof(Real),VE*Cabeza.NumFilas,fm);
   else
      /*Cargamos la trayectoria suponiendo guadada por columnas*/
      for(n=1, ta=Ptred->TraDese; n<=VE; n++, ta++)
  for(v=1, t=ta; v<=Cabeza.NumFilas; v++, t+=VE)
     fread((void*)t,sizeof(Real),1,fm);

   fclose(fm);

}

void DefineSalida(red,esnueva)
```

```
TRed red;
TLogico esnueva;
{
   TUnidad *PtU;
   Real Rtip[VS];
   SinSigno i,j,IUni,FTip[VS];

   if(esnueva)
   {
      printf("\nRango tipico para la variable de salida");
      EntraRango(Rtip);
      printf("\nFuncion tipica de salida");
      EntraFuncion(FTip);

      for(i=0, PtU=red.PtUni; i<red.N; i++, PtU++)
      {
   memcpy((void*)PtU->RanSal,(void*)Rtip,VS*sizeof(Real));
   memcpy((void*)PtU->FunSal,(void*)FTip,VS*sizeof(SinSigno));
      }
   }

   do
   {
      printf("\nUnid:  ");
      printf("\tRango de la salida:");
      printf("\tFunc. Activ:");

      for(i=0, PtU=red.PtUni; i<=red.N; i++, PtU++)
      {
   printf("\n %4hu ",i,PtU->L);
   for(j=0; j<VE ;j++)
      printf("\t %lG",PtU->RanSal[j]);
   printf("\n      ");
   for(j=0; j<VE ;j++)
      switch(PtU->FunSal[j])
      {
      case TH:
         printf("\t TanH"); break;
      case SG:
         printf("\t Sigm"); break;
      case LI:
         printf("\t Lin "); break;
      }
      }
      printf("\nEntra numero de unidad a modificar(>N para terminar):");
      scanf("%hu",&IUni);
      if (IUni<=red.N)
      {
   EntraRango((red.PtUni+IUni-1)->RanSal);
   EntraFuncion((red.PtUni+IUni-1)->FunSal);
      }
   } while(IUni<=red.N);
}

void MuesTraDes(red)
TRed red;
{
   Real *Ptd;
   SinSigno i,j;

   printf("\nLa trayectoria almacenada es:");
   for(i=1, Ptd=red.TraDese; i<=((size_t)*(red.TraDese-1))/VE; i++)
   {
      printf("\n");
      for(j=1; j<=VE; j++, Ptd++)
   printf("\t%lE",*Ptd);
   }
}

int Inicializa(PtRed)
TRed *PtRed;
{
   char c1;
   TLogico Esnueva;

   /* funcion para definir las caracteristicas de la red
      y obtener la memoria */

   printf("\nDefinicion de las caracteristicas de la red.\n");
   Esnueva= !PtRed->N;

   if(Esnueva && !CreaEstructura(PtRed))
      return 0;
   do
   {
      PresentaCapas(*PtRed);
      printf("\n Deseas mantener esta estructura(s/n)");
      while((c1=getchar())!='s' && c1!='n');
      if (c1=='n')
      {
   Libera(PtRed);
   if(!CreaEstructura(PtRed))
      return 0;
      }
   }while( c1=='n' );


   /*Rangos y funciones de salida*/
   DefineSalida(*PtRed,Esnueva);


   /*Para introducir la trayectoria deseada*/
   if(PtRed->TraDese /*!=(Real*)void*/ )
   {
      printf("\nExiste Trayectoria Deseada");
      MuesTraDes(*PtRed);
      printf("\nConservar/Cargar/No pesar(c/s/n)");
      while((c1=(getchar()|0x20))!='s' && c1!='n' && c1!='c');
   }
   else
   {
      printf("\nSe pesara la trayectoria deseada(s/n)");
      while((c1=(getchar()|0x20))!='s' && c1!='n');
   }
   if(c1=='s')
      CargaTraDesMat(PtRed);
```

```
    else
       if(c1=='n' && PtRed->TraDese)
       {
    free((void*)(PtRed->TraDese-1));
    PtRed->TraDese= /*(Real*)void*/ 0;
       }


    /*Inicializacion de los pesos*/
    printf("\nPesos: Actuales/Inicializar/Cargar?(a/i/c)");
         while((c1=getchar())!='a' && c1!='i' && c1!='c');
    if( c1=='i')
      IniciaPesos(*PtRed);
    else if(c1=='c')
       CargaPesM(PtRed);

    PtRed->NumItera=0;
    return 1; /* terminacion correcta */

}

void MusAlgo(PtRed)
TRed *PtRed;
{
   char c;

   do
   {
      /*Algoritmo de entrenamiento*/
      printf("\nAlgoritmo de entrenamiento");
      printf("\n\tG- Numero de puntos para el gradiente:");
      printf("%lu",PtRed->NumGrad);

      printf("\n\tEscala en el calculo del gradiente:");
      printf("\n\t\t %c N-No escala",
      PtRed->AlgDir & SINESCALA?'*':' ');
      printf("\n\t\t %c K-Escala por red",
      PtRed->AlgDir & ESCALARED?'*':' ');
      printf("\n\t\t %c T-Escala a todas las redes",
      PtRed->AlgDir & ESCALATOT?'*':' ');
      if(PtRed->AlgDir & (ESCALARED|ESCALATOT))
   printf("\n\t\t\tE-Escala de la primera capa= %lG"
      ,PtRed->PetMin);
      printf("\n\tCalculo de la direccion de busqueda:");
      printf("\n\t\t %c 1-Direccion del gradiente",
      PtRed->AlgDir & DIRGRA?'*':' ');

      printf("\n\t\t %c 2-Conjugado Fletcher",
      PtRed->AlgDir & DIRFLE?'*':' ');
      if(PtRed->AlgDir & DIRFLE)
      {
   printf("\n\t\t\tC -Cuenta de reseteo:%lu",PtRed->NumReset);
   if(!PtRed->NumReset)
      printf("\n\t\t\tR -Rango de proximidad= %lG",PtRed->Rango);
      }
      printf("\n\t\t %c 3-Conjugado Polac-Ribbier (autoreseteo)",
      PtRed->AlgDir & DIRPOL?'*':' ');

      printf("\n\tAlgoritmo de Busqueda lineal:");

      printf("\n\t\t %c 4-Paso Fijo.",
       PtRed->AlgBus & BFIJA?'*':' ');
      if(PtRed->AlgBus & BFIJA)
         printf("\n\t\t\tA -Alfa: %lG",PtRed->Alfa);

      printf("\n\t\t %c u-Un paso.(aumneto c1 dism c2)",
       PtRed->AlgBus & BSimple?'*':' ');

      printf("\n\t\t %c 5-Parabolica sin peta",
      PtRed->AlgBus & BsinPET?'*':' ');

      printf("\n\t\t %c 6-Parabolica con peta",
       PtRed->AlgBus & BconPET?'*':' ');
      printf("\n\t\t\tP -Peta Minima= %lG",PtRed->PetMin);
      printf("\n\t\t %c s-Parabolica hasta saturacion",
       PtRed->AlgBus & Bsatura?'*':' ');

      if(!(PtRed->AlgBus & BFIJA))
      {
   printf("\n\tPaso inicial de busqueda:");
   printf("\n\t\t %c F-Fijo (=ALfa)",
         PtRed->AlgBus & BIfija?'*':' ');
   if(PtRed->AlgBus & BIfija)
      printf("\n\t\t\tA -Alfa: %lG",PtRed->Alfa);
   printf("\n\t\t %c V-Alfa previo",
         PtRed->AlgBus & BIalp?'*':' ');
   printf("\n\t\t %c M-t0 minimo",
         PtRed->AlgBus & BImin?'*':' ');

   printf("\n\tMetodo de terminacion");
   printf("\n\t\t %c 7-Al primer Fo",
            PtRed->AlgTer & FPRIMER?'*':' ');
   printf("\n\t\t %c 8-Cuando Fo<Fa",
            PtRed->AlgTer & FMFA?'*':' ');
   printf("\n\t\t %c 9-Cuando Fo<=Fb",
            PtRed->AlgTer & FMFB?'*':' ');
      }


      printf("\nMarca opcion a modificar (0 termina)");
      do{ c=getchar(); } while(c<' '); /*Evita caracteres especiales*/
      switch (c|0x20) /*Pasa a minusculas*/
      {
      case 'n': PtRed->AlgDir=(PtRed->AlgDir & 0x0F)|SINESCALA; break;
      case 'k': PtRed->AlgDir=(PtRed->AlgDir & 0x0F)|ESCALARED; break;
      case 't': PtRed->AlgDir=(PtRed->AlgDir & 0x0F)|ESCALATOT; break;

      case '1': PtRed->AlgDir=(PtRed->AlgDir & 0xF0)|DIRGRA; break;
      case '2': PtRed->AlgDir=(PtRed->AlgDir & 0xF0)|DIRFLE; break;
      case '3': PtRed->AlgDir=(PtRed->AlgDir & 0xF0)|DIRPOL; break;

      case '4': PtRed->AlgBus=(PtRed->AlgBus & 0x70)|BFIJA; break;
      case 'u': PtRed->AlgBus=(PtRed->AlgBus & 0x70)|BSimple; break;
      case '5': PtRed->AlgBus=(PtRed->AlgBus & 0x70)|BsinPET; break;
      case '6': PtRed->AlgBus=(PtRed->AlgBus & 0x70)|BconPET; break;
      case 's': PtRed->AlgBus=(PtRed->AlgBus & 0x70)|Bsatura; break;
```

```
      case 'f': PtRed->AlgBus=(PtRed->AlgBus & 0x0F)|BIfija; break;
      case 'v': PtRed->AlgBus=(PtRed->AlgBus & 0x0F)|BIalp; break;
      case 'm': PtRed->AlgBus=(PtRed->AlgBus & 0x0F)|BImin; break;

      case '7': PtRed->AlgTer=FPRIMER; break;
      case '8': PtRed->AlgTer=FMFA; break;
      case '9': PtRed->AlgTer=FMFB; break;

      case 'g': printf("\nNumero de gradientes? =");
         scanf("%lu",&PtRed->NumGrad);
         break;
      case 'c': printf("\nCuenta de Reseteo? =");
         scanf("%lu",&PtRed->NumReset);
         break;
      case 'r': printf("\nRango de proximidad? =");
         scanf("%lG",&PtRed->Rango);
         break;
      case 'e': printf("\nEscala primera capa? =");
         scanf("%lG",&PtRed->PetMin);
         break;
      case 'a': printf("\nAlfa? =");
         scanf("%lG",&PtRed->Alfa);
         break;
      case 'p': printf("\nPeta Minima? =");
         scanf("%lG",&PtRed->PetMin);
         break;
      }

   } while (c!='0');
}



void MuestraParametros(Red)
TRed Red;
{
   SinSigno i;

   /* Mostrar la actual zona entrenamiento */
   printf("\nLa actual zona de entrenamiento es:");
   for(i=1; i<=VE; i++)
   {
      printf("\nE- Para Variable de entrada X(%hu):",i);
      printf(" %11G --> %11G ",Red.REntrena.Xmin[i-1],
            Red.REntrena.Xmax[i-1]);
   }

#ifndef  Met1
   printf("\nC- La region de comandos iniciales es:");
   for(i=1; i<=VC; i++)
   {
      printf("\nPara Variable comando X(%hu):",i);
      printf(" %11G --> %11G ",Red.ComIni.Cmin[i-1],
            Red.ComIni.Cmax[i-1]);
   }
#endif  /*De la region de comandos. No Met1*/

   printf("\nConstantes son");
   printf("\n1-   c1= %lG ",Red.c1);
   printf("\n2-   c2= %lG ",Red.c2);
   printf("\n3-   c3= %lG ",Red.c3);
   printf("\nPara la formula:");
   printf(" c3/(1+c2*t)+c1*t    (t no. entrenamientos)");
   printf("\nO factores de aumento, disminucion y alfa inicial");
   printf("\n  Se llevan realizados %lu entrenamientos",Red.NumItera);

}


void EntraParametros(PtRed)
TRed *PtRed;
{
   Real xi,xf;
   SinSigno i;
   TLogico Correcto;
   char c;

   do
   {
      MuestraParametros(*PtRed);
      printf("\n Parametro a modificar (0 para salir)");
      while((c=getchar())<' ');
      switch (c|0x20)
      {
      case 'e':
  printf("\nPara la REGION de ENTRENAMIENTO:");
  do
  {
     Correcto= VERDAD;
     printf("\n Entra nuevas coordenadas:");
     for(i=1; i<=VE; i++)
     {
        printf("\nPara X(%hu):\n Valor minimo=",i);
        scanf("%lg",&xi);
        printf(" Valor maximo=");
        scanf("%lg",&xf);
        if  (xi>xf)
        {
       printf("\n ERROR: Valor minimo ha de ser <= que maximo");
           Correcto=FALSO;
           break;
        }
        PtRed->REntrena.Xmin[i-1]=xi;
        PtRed->REntrena.Xmax[i-1]=xf;
     }
  } while(! Correcto);
  break;

#ifndef  Met1
      case 'c':
  printf("\nValores de los comandos U0:");
  do
  {
     Correcto= VERDAD;
```

```
      printf("\n Entra nuevas extremos:");
      for(i=1; i<=VC; i++)
      {
         printf("\nPara U0(%hu):\n Valor minimo=",i);
         scanf("%lg",&xi);
         printf(" Valor maximo=");
         scanf("%lg",&xf);
         if  (xi>xf)
         {
    printf("\n ERROR: Valor minimo ha de ser <= que maximo");
            Correcto=FALSO;
            break;
         }
         PtRed->ComIni.Cmin[i-1]=xi;
         PtRed->ComIni.Cmax[i-1]=xf;
      }
   } while(! Correcto);
   break;
#endif  DE valores comandos. No Met1
      case '1':
   printf("   c1="); scanf("%lg",&PtRed->c1);
   break;
      case '2':
   printf("   c2="); scanf("%lg",&PtRed->c2);
   break;
      case '3':
   printf("   c3="); scanf("%lg",&PtRed->c3);
   break;
      } /*cerramos Switch*/
   } while (c!='0');
}
```

A.2.1.9 muescom.c

```
#include <stdio.h>
#include <string.h>
#include "defred.h"


void SalEstMat(Red,Fich,NomB,SalP,SalG,SalY,SalZ,SalD)
TRed Red;
FILE *Fich;
char *NomB;
TLogico SalP,SalG,SalY,SalZ,SalD;
{
   CabezaMat Cabeza;
   TUnidad *UAct;
   SinSigno i,MaxP;
   char NomV[30];

   Cabeza.tipo=1000;  /* recorre por columnas */
   Cabeza.HayImag=0;
   Cabeza.NumColum=Red.N;   /*tantas columnas como etapas*/


   /* guardar informacion de la red segun indiquen los flags
      Sera una matriz con tantas columnas como unidades,
   y tantas filas como pesos tengan las unidades => sup.
   unidades iguales*/

   UAct=Red.PtUni;

   MaxP=UAct->TotPes;
   for(i=1, UAct++; (i<Red.N)&&(MaxP==UAct->TotPes); i++, UAct++);
   if(i<Red.N)
   {
      printf("\nEsta rutina esta preparada solo para redes iguales");
      printf("\nNo se ha salvado estado");
      return;
   }

   /*Para los pesos*/
   if(SalP)
   {
      Cabeza.NumFilas=Red.PtUni->TotPes;  /*Tantas Filas como pesos*/
      strcpy(NomV,NomB);
      strcat(NomV,"P");
      Cabeza.TamNom=strlen(NomV)+1;

      fwrite((void*)&Cabeza,sizeof(CabezaMat),1,Fich);
      fwrite((void*)NomV,sizeof(char),(size_t)Cabeza.TamNom,Fich);

      for(i=0, UAct=Red.PtUni; i<Red.N; i++,UAct++)
      fwrite((void *)UAct->Wa,sizeof(Real),(size_t)UAct->TotPes,Fich);
   }

   /*Para los gradientes*/
   if(SalG)
   {
      Cabeza.NumFilas=Red.PtUni->TotPes;  /*Tantas Filas como pesos*/
      strcpy(NomV,NomB);
      strcat(NomV,"G");
      Cabeza.TamNom=strlen(NomV)+1;

      fwrite((void*)&Cabeza,sizeof(CabezaMat),1,Fich);
      fwrite((void*)NomV,sizeof(char),(size_t)Cabeza.TamNom,Fich);

      for(i=0, UAct=Red.PtUni; i<Red.N; i++,UAct++)
      fwrite((void *)UAct->Grad,sizeof(Real),(size_t)UAct->TotPes,Fich);
   }

   /*Para los Y*/
   if(SalY)
   {
      Cabeza.NumFilas=Red.PtUni->TotNeu;  /*Tantas Filas como Neuronas*/
      strcpy(NomV,NomB);
      strcat(NomV,"Y");
      Cabeza.TamNom=strlen(NomV)+1;

      fwrite((void*)&Cabeza,sizeof(CabezaMat),1,Fich);
      fwrite((void*)NomV,sizeof(char),(size_t)Cabeza.TamNom,Fich);

      for(i=0, UAct=Red.PtUni; i<Red.N; i++,UAct++)
```

```
            fwrite((void *)UAct->Y,sizeof(Real),(size_t)UAct->TotNeu,Fich);
        }

        /*Para los Z*/
        if(SalZ)
        {
            Cabeza.NumFilas=Red.PtUni->TotNeu;  /*Tantas Filas como Neuronas*/
            strcpy(NomV,NomB);
            strcat(NomV,"Z");
            Cabeza.TamNom=strlen(NomV)+1;

            fwrite((void*)&Cabeza,sizeof(CabezaMat),1,Fich);
            fwrite((void*)NomV,sizeof(char),(size_t)Cabeza.TamNom,Fich);

            for(i=0, UAct=Red.PtUni; i<Red.N; i++,UAct++)
            fwrite((void *)UAct->Z,sizeof(Real),(size_t)UAct->TotNeu,Fich);
        }

        /*Para los Deltas*/
        if(SalD)
        {
            Cabeza.NumFilas=Red.PtUni->TotNeu;  /*Tantas Filas como Neuronas*/
            strcpy(NomV,NomB);
            strcat(NomV,"D");
            Cabeza.TamNom=strlen(NomV)+1;

            fwrite((void*)&Cabeza,sizeof(CabezaMat),1,Fich);
            fwrite((void*)NomV,sizeof(char),(size_t)Cabeza.TamNom,Fich);

            for(i=0, UAct=Red.PtUni; i<Red.N; i++,UAct++)
        fwrite((void *)UAct->Delta,sizeof(Real),(size_t)UAct->TotNeu,Fich);
        }

    } /* de Guarda estado  Mat */


    int SalvaEstado(Red,NomFS)
    TRed Red;
    char *NomFS;
    {
       char c,NomFil[25];
       TLogico Salir,SalP,SalG,SalY,SalZ,SalD;
       FILE *Fich;

       printf("\n Salvar distinta datos de la red en estado actual.");
       if (*NomFS)
       {
          printf("\nQuieres salvar en fichero '%s'(.mat) ?(s/n)",NomFS);
          while( (c=(getchar() | 0x20))!='s' &&  c!='n') ;
          if (c=='n')
       NomFS[0]=0;
       }
       do
       {
          if (! *NomFS )
          {
       printf("\nNombre del fichero(sin .mat):");
       scanf("%s",NomFS);
          }
          Salir=VERDAD;
          if(*NomFS)
          {
       strcpy(NomFil,NomFS);
       strcat(NomFil,".mat");
              if (Fich=fopen(NomFil,"w"))
       {
          /*Turno de Preguntas*/
          printf("\nSalvar Pesos?(s/n)");
          while( (c=(getchar() | 0x20))!='s' &&  c!='n') ;
          SalP=(c=='s');
          printf("\nSalvar Gradiente?(s/n)");
          while( (c=(getchar() | 0x20))!='s' &&  c!='n') ;
          SalG=(c=='s');
          printf("\nSalvar Y?(s/n)");
          while( (c=(getchar() | 0x20))!='s' &&  c!='n') ;
          SalY=(c=='s');
          printf("\nSalvar Z?(s/n)");
          while( (c=(getchar() | 0x20))!='s' &&  c!='n') ;
          SalZ=(c=='s');
          printf("\nSalvar Delta?(s/n)");
          while( (c=(getchar() | 0x20))!='s' &&  c!='n') ;
          SalD=(c=='s');
          SalEstMat(Red,Fich,NomFS,SalP,SalG,SalY,SalZ,SalD);
          fclose(Fich);
       }
       else
       {
          printf("No puedo abrir ese fichero.");
          Salir=FALSO;
       }
          }
       }
       while(!Salir);
    }
```

A.2.1.10 prigen.c

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include "defred.h"

#define MAXPTINI 20


int Salva(/*Red,NomFS*/);
int Inicializa(/*Red*/);
int SalvaRed(/*Red,NomF*/);
void MusAlgo();
void EntraParametros(/*&Red*/);
void MuestraParametros(/*Red*/);
void Entrena();
```

```
/***** Variables Globales  **********/
/*Fichero de errores*/
FILE *error;
volatile TLogico terminar=FALSO;
/*********************************/


int TrataSig(nums,tipo,strusig)
int nums;
int tipo;
struct sigcontext *strusig;
{
   /* Cuando se detecta la seÀal se conmuta la variable terminar
   en el bucle de entrenamiento se comprueba esta seÀal */
   terminar=VERDAD;

   /* vuelve a preparar la captura de la seÀal */
   signal(SIGUSR1,(void *)TrataSig);
}


int main(argc,argv)
int argc;
char *argv[];
{
   TRed Red;
   char c;
   int ok;
   TLogico EstaSalvado,PuedeSalir;
   char NomRed[25],NomSeg[28],NomPes[28];
   /*Para poder determinar si existe stdout con ctermid() */
   char term[L_ctermid];

   /* prepara captura seÀal USR1 */
   signal(SIGUSR1,(void *)TrataSig);

   NomPes[0]='\0';
   NomRed[0]='\0'; /* nombre vacio */

   /*Inicializacion de los valores por defecto*/
   Red.N=0;  /* esto indica que por el momento no existe red */
   Red.AlgDir=DIRFLE|SINESCALA;  /*inicializaciones del algoritmo*/
   Red.AlgBus=BFIJA;
   Red.AlgTer=FMFA;
   Red.NumGrad=1;
   Red.NumReset=0;
   Red.Rango=0.01;
   Red.PetMin=1e-8;
   Red.Alfa=0.01;
   Red.TraDese=0;
   Red.REntrena.Xmin[0]=0;  Red.REntrena.Xmax[0]=0;
   Red.REntrena.Xmin[1]=0;  Red.REntrena.Xmax[1]=0;

   EstaSalvado=FALSO;
   printf("\nComienza el programa\n");

   /*Carga la red o la inicializa*/
   if(argc>1)
   {
      strcpy(NomRed,argv[1]);
      if(! CargaRed(&Red,NomRed))
      {
         printf("\nNo puedo habrir fichero '%s' o no tengo memoria\n",
      NomRed);
         return 1;
      }
      EstaSalvado=VERDAD;
   }
   else
     if(! Inicializa(&Red))
     {
       printf("\nNo puedo obtener memoria para la red");
       return 1;
     }

   /*Determina la salida de error*/
   printf("\nEntra nombre de la salida de error,[stderr]");
   scanf("%s",NomPes);
   if(!*NomPes)
      error=stderr;
   else if(!(error=fopen(NomPes,"w")))
   {
      printf("\nNo puedo abrir la salida de error");
      return 1;
   }

   /* bucle principla del programa */
   do
   {
      PuedeSalir=FALSO;
      printf("\n Entrenamientos realizados: %lu\n",Red.NumItera);

      printf("\nElige opcion:");
      printf("\n     1-Etrenar");
      printf("\n     2-Calcular");
      printf("\n     3-Guardar");
      printf("\n     4-Inizalizar");
      printf("\n     5-Serie de Entrenamientos");
      printf("\n     7-Salva datos a MAT");
      printf("\n     e-Fijar el numero de entrenamiento");
      printf("\n     a-Fijar el tipo de algoritmo");
      printf("\n     p-Modificar parametros");
      printf("\n\n     9-Terminar\n");
      while((c=getchar())<' ');
      switch (c | 0x20)
      {
         case '1':
            Entrena(&Red);
            EstaSalvado=FALSO;
            /* copia de seguridad */
            strcpy(NomSeg,NomRed); strcat(NomSeg,".sg");
            SalvaRed(Red,NomSeg);
      ctermid(term);if(!*term) exit(1);
            break;

         case '2':
```

```
                if(!CalculaTrayecN(Red))
                   printf("\nProblemas al hacer trayectorias");
                break;

             case '3':
                ok=Salva(Red,NomRed);
                if (!ok)
                   printf("\nProblemas salvando la red");
                else
                   EstaSalvado=VERDAD;
                break;

             case '4':
                if (! Inicializa(&Red))
                {
                   printf("\nNo puedo obtener memoria para la red");
                   return 1;
                }
                else
                   EstaSalvado=FALSO;
                break;
             case '5':
                EntreSerie(&Red);
                EstaSalvado=FALSO;
                /* copia de seguridad */
                strcpy(NomSeg,NomRed); strcat(NomSeg,".sg");
                SalvaRed(Red,NomSeg);
          ctermid(term);
/*             if(!*term) exit(1);
          else*/
             fprintf(error,"\nEl terminal es >%s<",term);
                break;

            case '7':
                SalvaEstado(Red,NomPes);
                break;

      case 'e':
          printf("\nNuevo valor para el contador de entrenamiento?");
          scanf("%lu",&Red.NumItera);
          break;

      case 'a': /*Parametros del algoritmo*/
          MusAlgo(&Red);
          break;

      case 'p':
          EntraParametros(&Red);
          break;

             case '9':
                if( EstaSalvado)
                   PuedeSalir=VERDAD;
                else
                {
                   printf("<PULSA>"); getchar();
                   printf("\nLa red ha sido modificada y NO salvada");
                   printf("\nEstas seguro que deseas terminar(s/[n])");
                   if ((c=getchar()|0x20)=='s')
                      PuedeSalir=VERDAD;
                }
          }
       }
    while(!PuedeSalir);
/*  printf("\n\n"); para poder acabar si no existe terminal*/
    if(error!=stderr)
       fclose(error);
    return 0;
}
```

A.2.1.11 tragen.c

```
#include "defred.h"
/*#include "decfunc.h"*/
#include <stdio.h>
#include <string.h>


#define MAXPTINI 20

/*Funciones exteriores usadas*/
void MuestraParametros();
void CalculaRed();
Real H();
Real HN();
SinSigno PtosPro();

Real zeros[VC];  /* variable a cero */

void CalculaRed(/*Red,Xini,Uo*/);
void MuestraParametros(/*Red*/);

MuesPtos(Xo,k)
Real *Xo;
SinSigno k;
{
   Real *xa;
   SinSigno i,j;
   xa=Xo;
   for(i=1; i<=k; i++)
   {
      printf("\nPto %hu: X(",i);
      for(j=1; j<=VE; j++)
      {
   printf("%le,",*xa);
   xa++;
      }
#ifndef  Met1
      printf(")U(");
      for(j=1; j<=VC; j++)
      {
   printf("%le,",*xa);
   xa++;
```

```
      }
#endif
      printf(")");
   }
   printf("\n");
}


unsigned EntraIni(Red,Xo)
TRed Red;
Real *Xo;
{
   Real *xa;
   SinSigno i,k;
   TLogico NoSeg;

   /* Presentacion de informacion de region */
   MuestraParametros(Red);

   printf("\nEntra los puntos iniciales(hasta %u)",MAXPTINI);
   printf("\nEntra un carcter para terminar");
   NoSeg=FALSO;
   for(k=0, xa=Xo, NoSeg=FALSO;!NoSeg && k<MAXPTINI; )
   {
      printf("\n Punto inicial %u:\n",k+1);
      for(i=1;!NoSeg && i<=VE;i++, xa++)
      {
         printf("X[%hu]=",i);
         NoSeg=!scanf("%lf",xa);
      }
#ifndef _Met1
      for(i=1;!NoSeg && i<=VE;i++, xa++)
      {
   printf("Uo[%hu]=",i);
   NoSeg=!scanf("%lf",xa);
      }
#endif
      if(! NoSeg)
   k++;
   } /* del while k */
   /*Si no se ha entrado ninguno se toma los extremos de la region*/
   if (!k)
   {
      printf("\nTomando puntos promedio de la region");
      k=PtosPro(Red,Xo);
   }
   MuesPtos(Xo,k);
   return k;
}  /* de entra ini */


void PreparaTrayec(Red,Tray)
TRed Red;
TTrayec *Tray;
{
   /* Coloca las x y las consignas de cada etapa
     y calcula el costo */
   TUnidad *Uact;
   TTrayec *Ptr;
   Real CostoA;
   SinSigno u;

   CostoA=0.0;
   for(u=0, Uact=Red.PtUni, Ptr=Tray;
       u<Red.N;
       u++, Uact++, Ptr++)
   {
      memcpy((void*)Ptr->xa,(void*)Uact->Y,VE*sizeof(Real));
#ifdef _Met1
      memcpy((void*)Ptr->u,(void*)(Uact->u),VC*sizeof(Real));
      /*el tiempo ya esta al final*/
      CostoA+=H(Uact->Y,Uact->u,Red.TraDese,u);
/*en el metodo 2 el tiempo tambien esta al final*/
#else
    /*  memcpy((void*)Ptr->u,(void*)(Uact->u+1),(VC-1)*sizeof(Real));
      Ptr->u[VC-1]=Uact->u[0];   ponemos tiempo al final*/
      memcpy((void*)Ptr->u,(void*)(Uact->u),VC*sizeof(Real));
      CostoA+=H(Uact->Y,(Uact+1)->u,Red.TraDese,u);
#endif */
      Ptr->costo=CostoA;
   }
   /* caso de ultima unidad (N):
   - el costo se calcula con Hn */
   memcpy((void*)Ptr->xa,(void*)Uact->Y,VE*sizeof(Real));
#ifndef _Met1
 /*El tiempo ya esta al final
 memcpy((void*)Ptr->u,(void*)(Uact->u+1),(VC-1)*sizeof(Real));
   Ptr->u[VC-1]=Uact->u[0]; ponemos tiempo al final*/
   memcpy((void*)Ptr->u,(void*)(Uact->u),VC*sizeof(Real));
#else
   /*para el metodo 1 se desprecian ultimos comandos-> repetimos*/
   memcpy((void*)Ptr->u,(void*)(Ptr-1)->u,VC*sizeof(Real));
#endif
   Ptr->costo=CostoA+HN(Ptr->xa);
}

CompTra(Red,x0,tr,NumIni)
TRed Red;
Real *x0;
TTrayec *tr[];
SinSigno NumIni;
{
   SinSigno n;
   Real *xa;
#ifdef _Met1
   for(n=0, xa=x0; n<NumIni ;n++,xa+=VE)
   {
      CalculaRed(Red,xa);
      PreparaTrayec(Red,tr[n]);
   }
#else
   for(n=0, xa=x0; n<NumIni ;n++,xa+=VE+VC)
   {
      CalculaRed(Red,xa,xa+VE);
      PreparaTrayec(Red,tr[n]);
   }
```

```
#endif
}

int SalTraMat(Red,Trayec,NumTra,Fich,NomV,Ptdat,ndt)
TRed Red;
TTrayec *Trayec[MAXPTINI];
SinSigno NumTra;
FILE *Fich;
char *NomV;
Real *Ptdat;
SinSigno ndt;  /*Numero de filas de datos salvadas*/
{
   CabezaMat Cabeza;
   Real w[VC+VE+1];
   char Nomdat[20];

   SinSigno k;

   Cabeza.tipo=1100;  /* recorre por filas */
   Cabeza.HayImag=0;


   /* guardar todas las trayectorias en una matriz.
       - las primeras lineas indican el numero de puntos
     y los parametros.
       - las siguientes los VE+VC+1 valores  de cada punto */

   Cabeza.NumFilas=((Red.N+1)*NumTra)+NumTra*3;
   Cabeza.NumColum=VE+VC+1;
   Cabeza.TamNom=strlen(NomV)+1;

   fwrite((void *)&Cabeza,sizeof(CabezaMat),1,Fich);
   fwrite((void *)NomV,sizeof(char),(size_t)Cabeza.TamNom,Fich);
   for(k=0;k<NumTra;k++)
   {
      w[0]=Red.N+1;  /* indicamos el no. de puntos de la trayec*/
      w[1]=VE; w[2]=VC;
      /* en todas las columnas*/
      fwrite((void *)w,sizeof(Real),VC+VE+1,Fich);
      w[0]=Red.NumItera;
      w[1]=Red.Eta; /*Contendra factor conjugado*/
      w[2]=Red.c1;
      fwrite((void *)w,sizeof(Real),VC+VE+1,Fich);
      w[1]=Red.c3; /*Contendra ultimo alfaoptimo*/
      fwrite((void *)w,sizeof(Real),VC+VE+1,Fich);
                     /* hasta completar 3 lineas */
      /* escribimos los datos de la trayectoria */
      fwrite((void *)Trayec[k],sizeof(Real),
      (size_t)((Red.N+1)*(VC+VE+1)),Fich);
   }

   /*Salvamos los datos*/

   if(ndt)
   {
      Cabeza.NumFilas=ndt;
      Cabeza.NumColum=NDAT;
      strcpy(Nomdat,NomV);
      strcat(Nomdat,"DT");
      Cabeza.TamNom=strlen(Nomdat)+1;

      fwrite((void*)&Cabeza,sizeof(CabezaMat),1,Fich);
      fwrite((void*)Nomdat,sizeof(char),(size_t)Cabeza.TamNom,Fich);

      fwrite((void*)Ptdat,sizeof(Real),(size_t)(ndt*NDAT),Fich);
   }

   return 1; /*ya que no se controla que haya salido todo*/
} /* de Guarda Trayectoria en Mat */


int CalculaTrayecN(Red)
TRed Red;
{
/* Calcula y saca trayectorias */
   Real x0[MAXPTINI*(VE+VC)]; /*Reserva de sobra*/
   TTrayec *tr[MAXPTINI];
   SinSigno NumIni,n;
   char c,NomV[20],Nomf[20];
   FILE *FiMat;

   printf("\n Calculo de trayectoria.");


   NumIni=EntraIni(Red,x0);
   for(n=0; n<NumIni ;n++)
      if(!(tr[n]=malloc((Red.N+1)*sizeof(TTrayec))))
         return 0;
   CompTra(Red,x0,tr,NumIni);

   printf("\nQuieres sacar los datos a fichero MATLAB");
   printf(" o a pantalla?([m]/p)");
   while((c=getchar())!='m' && c!='p');
   if (c!='p')
   {
      do
      {
   printf("\n Nombre del fichero a salvar(sin .mat)?");
   scanf("%s",NomV);
   strcpy(Nomf,NomV);
   strcat(Nomf,".mat");
      }
      while(*Nomf && !(FiMat=fopen(Nomf,"w")));
      if(*Nomf)
      {
   SalTraMat(Red,tr,NumIni,FiMat,NomV,(Real*)0,0);
   fclose(FiMat);
      }
   }
   else
      printf("\nNo disponible presentacion en pantalla");
   /* liberamos memoria de trayectorias */
   for(n=0;n<NumIni;n++)
      free((void*)tr[n]);
   return 1;
}
```

A.2.2 Primer Sistema

A.2.2.1 p1l.h

```
/* funciones para la ecuacion de Van der Pol discretizada
   con differencia a delante para el primer metodo*/

#define A  1
#define B  0
#define C  1
#define D  0

/* ********** Funciones del problema ***************** */
#define X1k     x[0]
#define X2k     x[1]
#define Tk      u[1]
#define Uk      u[0]
#define X1k1    re[0]
#define X2k1    re[1]
#define X1N     Xn[0]
#define X2N     Xn[1]

void F(x,u,re)
Real *x,*u,*re;
{
   X1k1=X1k+Tk*X2k;
   X2k1=X2k+Tk*(A*X2k+B*X2k*X1k*X1k+D*X1k+C*Uk);
   return;
}

void ParcialXqF(x,u,q,re)
Real *x,*u,*re;
SinSigno q;
{
   if(q==1) /*parcial respecto a X1k*/
   {
      X1k1=1; X2k1=Tk*(D+2*B*X1k*X2k); return;
   }
   else if(q==2) /*parcial respecto a X2k*/
   {
      X1k1=Tk; X2k1=1+Tk*(A+B*X1k*X1k);  return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(x,u,q,re)
Real *x,*u,*re;
SinSigno q;
{
   if(q==1) /*respecto a uk*/
   {
      X1k1=0.0; X2k1=Tk*C; return;
   }
   else if(q==2)  /*respecto a Tk*/
   {
      re[0]=X2k; re[1]=(A*X2k+B*X2k*X1k*X1k+D*X1k+C*Uk);
   }
   else exit(3);  /* acceso incorrecto */
}

/*Funcion de costo lineal del tiempo*/
#define Vn  100
#define FESTA 100

Real H(x,u,TDes,NEta)
Real *x,*u,*TDes;
SinSigno NEta;
{
   Real costo, *td;
   SinSigno i;
   costo=Tk;  /*Costo por defecto*/
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
      if(!isinf(td[i]))
        costo+=FESTA*(x[i]-td[i])*(x[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(x,u,TDes,NEta,re)
Real *x,*u,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;
   if (!NEta || !TDes)
   {
      re[0]=0.0; re[1]=0.0; return;
   }
   else
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      re[i]=isinf(td[i])?0:FESTA*2*(x[i]-td[i]);
   }
}

void ParcialUdeH( x,u,re )
Real *x,*u,*re;
{
   re[0]=0.0; re[1]=1.0;  return;
}

Real HN( Xn )
Real *Xn;
{
   return Vn*(X1N*X1N+X2N*X2N);
}
```

```
void ParcialXnHn( Xn,Ln )
Real *Xn,*Ln;
{
   Ln[0]=2*Vn*X1N;
   Ln[1]=2*Vn*X2N;
   return;
}
```

A.2.2.2 p2la.h

```
/* funciones para la ecuacion de Van der Pol discretizada, mediante
   la discretizaciãn cl-sica, con differencia alante  */

#define A  1
#define B  0
#define C  1
#define D  0

#define VAL_P 1e-100
/* ********** Funciones del problema ***************** */

#define X1k  (xk[0])
#define X2k  (xk[1])
#define X1k1 (xk1[0])
#define X2k1 (xk1[1])
#define Tk   (uk[1])
#define Uk   (uk[0])
#define Tk1  (uk1[1])
#define Uk1  (uk1[0])
#define dUk1 (re[0])
#define dTk1 (re[1])

/** Discretizaciãn cl-sica **/
void F(xk,xk1,uk,uk1)
Real xk[VE],xk1[VE],uk[VC],uk1[VC];
{
   Tk1=(X1k1-X1k)/X2k;
   if(!Tk1)
      Tk1=VAL_P;
   Uk1=((X2k1-X2k)/Tk1-A*X2k-B*X2k*X1k*X1k-D*X1k)/C;
   return;
}

void ParcialXqF(xk,xk1,uk,uk1,q,re)
Real xk[VE],xk1[VE],uk[VC],uk1[VC],re[VC];
SinSigno q;
{
   if(q==1) /*Parcial F respecto a X1K */
   {
      dTk1=-1/X2k1;
      dUk1=(-dTk1*(X2k1-X2k)/(Tk1*Tk1)-2*B*X2k*X1k-D)/C;
      return;
   }
   else if(q==2) /*Parcial F respecto a X2K*/
   {
      dTk1=-(X1k1-X1k)/(X2k*X2k);
      dUk1=((X2k1-2*X2k)/(X2k*Tk1)-A-B*X1k*X1k)/C;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialX1qF(xk,xk1,uk,uk1,q,re)
Real xk[VE],xk1[VE],uk[VC],uk1[VC],re[VC];
SinSigno q;
{
   if(q==1) /*Parcial F respecto a X1K1*/
   {
      dTk1=1/X2k;
      dUk1=-(X2k1-X2k)/(C*Tk1*Tk1*X2k);
      return;
   }
   else if(q==2) /*Parcial F respecto a X2K1*/
   {
      dTk1=0;
      dUk1=1/(C*Tk1);
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(xk,xk1,uk,uk1,q,re)
Real xk[VE],xk1[VE],uk[VC],uk1[VC],re[VC];
SinSigno q;
{
   if(q==2) /* Parcial F respecto a Tk */
   {
      dTk1=0; dUk1=0;
      return;
   }
   else if(q==1)  /*Parcial de F respecto a Uk*/
   {
      dTk1=0; dUk1=0;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

/*Obtamos por funcion de coste derivable en todos los puntos*/

/*Funcion de coste que garantiza tiempos positivos ya que da costo
   mayor para los tiempos negativos en todas las situaciones, incluso
   para -1<t<0 (13/4/93)
   -Pesamos tambien el comando para evitar que salga de cierto rango
     (21/4/93)*/
/*(28/6/93) Como ahora garantizamos que los pesos iniciales no dan
   trayectorias con tiempos negativos podemos pesar estos mucho mas
   y tomar para los tiempos positivos y comandos una funcion lineal*/

#define FTPOS 10
#define Vn  1000
#define FTNEG  10000
```

```
#define RAN 5
#define  FACG 50
#define FESTA 100


Real H(xk,uk,TDes,NEta)
Real *xk,*uk,*TDes;
SinSigno NEta;
{
   Real costo, *td;
   SinSigno i;
   /*Costo por defecto*/
   costo=(Tk>=0)?FTPOS*Tk:1000000-Tk*FTNEG; /*No derivable*/
   if(Uk>RAN)
      costo+=FACG*(Uk-RAN);
   else if(Uk<-RAN)
      costo+=FACG*(-Uk-RAN);
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
  if(!isinf(xk[i]))
            costo+=FESTA*(xk[i]-td[i])*(xk[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(xk,uk,TDes,NEta,re)
Real *xk,*uk,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;
   if (!NEta || !TDes)
   {
      re[0]=0.0; re[1]=0.0; return;
   }
   else
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
         re[i]=isinf(xk[i])?0:FESTA*2*(xk[i]-td[i]);
   }
}


void ParcialUdeH( xk,uk,re )
Real xk[VE],uk[VC],re[VC];
{
   re[0]=(Tk>=0)?FTPOS:-FTNEG;
   if(Uk>RAN)
      re[1]=FACG;
   else if(Uk<-RAN)
      re[1]=-FACG;
   else
      re[1]=0.0;
   return;
}

Real HN( Xn )
Real Xn[VE];
{
   return Vn*(Xn[0]*Xn[0]+Xn[1]*Xn[1]);
}

void ParcialXnHn( Xn,Ln )
Real Xn[VE],Ln[VE];
{
   Ln[0]=2*Vn*Xn[0];
   Ln[1]=2*Vn*Xn[1];
   return;
}
```

A.2.2.3 p2bpa.h

```
/* funciones para la ecuacion de Van der Pol discretizada, mediante
   la discretizaciãn cl-sica, con differencia alante  */

#define A  1
#define B  0
#define C  1
#define D  0

#define VAL_P 1e-100
/* ********** Funciones del problema ***************** */

#define X1k  (xk[0])
#define X2k  (xk[1])
#define X1k1 (xk1[0])
#define X2k1 (xk1[1])
#define Tk   (uk[1])
#define Uk   (uk[0])
#define Tk1  (uk1[1])
#define Uk1  (uk1[0])
#define dUk1 (re[0])
#define dTk1 (re[1])

/*disc bpf*/
void F(xk,xk1,uk,uk1)
Real xk[VE],xk1[VE],uk[VC],uk1[VC];
{
   Tk1=2*(X1k1-X1k)/(X2k+X2k1);
   Uk1=( -C*Uk
   +2*(X2k1-X2k)/Tk1
   -A*(X2k+X2k1)
   -B*(X2k*X1k*X1k+X2k1*X1k1*X1k1)
   -D*(X1k+X1k1))/C;
   return;
}

void ParcialXqF(xk,xk1,uk,uk1,q,re)
Real xk[VE],xk1[VE],uk[VC],uk1[VC],re[VC];
```

```
SinSigno q;
{
   if(q==1) /*Parcial F respecto a X1k */
   {
      re[0]=-2/(X2k1+X2k);
      re[1]=(4*(X2k1-X2k)/(X2k+X2k1)/Tk1/Tk1
         -B*2*X2k*X1k
         -D)/C;
      return;
   }
   else if(q==2) /*Parcial F respecto a X2k*/
   {
      re[0]=-Tk1/(X2k1+X2k);
      re[1]=(-4*X2k/(X2k1+X2k)/Tk1
         -B*X1k*X1k
         -A)/C;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialX1qF(xk,xk1,uk,uk1,q,re)
Real xk[VE],xk1[VE],uk[VC],uk1[VC],re[VC];
SinSigno q;
{
   if(q==1) /*Parcial F respecto a X1k1*/
   {
      re[0]=2/(X2k1+X2k);
      re[1]=(-4*(X2k1-X2k)/(X2k+X2k1)/Tk1/Tk1
         -2*B*X2k1*X1k1
         -D)/C;
      return;
   }
   else if(q==2) /*Parcial F respecto a X2k1*/
   {
      re[0]=-Tk1/(X2k1+X2k);
      re[1]=(4*X2k/(X2k1+X2k)/Tk1
                -A
        -B*X1k1*X1k1)/C;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(xk,xk1,uk,uk1,q,re)
Real xk[VE],xk1[VE],uk[VC],uk1[VC],re[VC];
SinSigno q;
{
   if(q==2) /* Parcial F respecto a Tk */
   {
      re[0]=0;
      re[1]=0;
      return;
   }
   else if(q==1)  /*Parcial de F respecto a Uk*/
   {
      re[0]=0;
      re[1]=-1;
   }
   else exit(3);  /* acceso incorrecto */
}

/*Obtamos por funcion de coste derivable en todos los puntos*/

/*Funcion de coste que garantiza tiempos positivos ya que da costo
   mayor para los tiempos negativos en todas las situaciones, incluso
   para -1<t<0 (13/4/93)
   -Pesamos tambien el comando para evitar que salga de cierto rango
     (21/4/93)*/
/*(28/6/93) Como ahora garantizamos que los pesos iniciales no dan
   trayectorias con tiempos negativos podemos pesar estos mucho mas
   y tomar para los tiempos positivos y comandos una funcion lineal*/

#define FTPOS 10
#define Vn  1000
#define FTNEG  10000
#define RAN 5
#define  FACG 50
#define FESTA 100


Real H(xk,uk,TDes,NEta)
Real *xk,*uk,*TDes;
SinSigno NEta;
{
   Real costo, *td;
   SinSigno i;
   /*Costo por defecto*/
   costo=(Tk>=0)?FTPOS*Tk:1000000-Tk*FTNEG; /*No derivable*/
   if(Uk>RAN)
      costo+=FACG*(Uk-RAN);
   else if(Uk<-RAN)
      costo+=FACG*(-Uk-RAN);
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
  if(!isinf(xk[i]))
            costo+=FESTA*(xk[i]-td[i])*(xk[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(xk,uk,TDes,NEta,re)
Real *xk,*uk,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;
   if (!NEta || !TDes)
   {
      re[0]=0.0; re[1]=0.0; return;
   }
   else
   {
```

```
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
         re[i]=isinf(xk[i])?0:FESTA*2*(xk[i]-td[i]);
   }
}

void ParcialUdeH( xk,uk,re )
Real xk[VE],uk[VC],re[VC];
{
   re[0]=(Tk>=0)?FTPOS:-FTNEG;
   if(Uk>RAN)
      re[1]=FACG;
   else if(Uk<-RAN)
      re[1]=-FACG;
   else
      re[1]=0.0;
   return;
}

Real HN( Xn )
Real Xn[VE];
{
   return Vn*(Xn[0]*Xn[0]+Xn[1]*Xn[1]);
}

void ParcialXnHn( Xn,Ln )
Real Xn[VE],Ln[VE];
{
   Ln[0]=2*Vn*Xn[0];
   Ln[1]=2*Vn*Xn[1];
   return;
}
```

A.2.3 Segundo Sistema

A.2.3.1 p1l3l.h

```
/*Sistema movil HILARE (IEEE Transactions on Automatic Control,
vol 39, no.1, pag 219 */
/*Con variable x3N libre -> direccion final libre*/

/* ********** Constantes del problema *************** */

#define X1D      0.0     /*Valores del punto final*/
#define X2D      0.0
#define  X3D     0.0

#define A        0.01

#define SQR(x) (x)*(x)

/* ********** Funciones del problema ***************** */
#define X1k      x[0]
#define X2k      x[1]
#define X3k      x[2]
#define U1k      u[0]
#define U2k      u[1]
#define Tk       u[2]
#define X1k1     re[0]
#define X2k1     re[1]
#define X3k1     re[2]
#define X1N      Xn[0]
#define X2N      Xn[1]
#define X3N      Xn[2]


void F(x,u,re)
Real *x,*u,*re;
{
   X1k1=X1k+Tk*cos(X3k)*U1k;
   X2k1=X2k+Tk*(A+sin(X3k)*U1k);
   X3k1=X3k+Tk*U2k;
   return;
}

void ParcialXqF(x,u,q,re)
Real *x,*u,*re;
SinSigno q;
{
   if(q==1)  /*respecto a X1k*/
   {
      X1k1=1;
      X2k1=0;
      X3k1=0;
      return;
   }
   else if(q==2)  /*Respecto a X2k */
   {
      X1k1=0;
      X2k1=1;
      X3k1=0;
      return;
   }
   else if (q==3)  /*respecto a X3k*/
   {
      X1k1=-Tk*sin(X3k)*U1k;
      X2k1=Tk*cos(X3k)*U1k;
      X3k1=1;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(x,u,q,re)
Real *x,*u,*re;
SinSigno q;
```

```
{
   if(q==1)  /*Respecto a U1k*/
   {
      X1k1=Tk*cos(X3k);
      X2k1=Tk*sin(X3k);
      X3k1=0.0; return;
   }
   else if(q==2) /*Respecto a U2k*/
   {
      X1k1=0.0; X2k1=0.0; X3k1=Tk; return;
   }
   else if(q==3) /*Respecto a Tk*/
   {
      X1k1=cos(X3k)*U1k;
      X2k1=A+sin(X3k)*U1k;
      X3k1=U2k;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

#define Vn  10000
#define VARNEG 100   /*Pendiente var. est. negativas*/
#define VARG   100   /*Pendiente var. est. grandes*/
#define MAXH1  10
#define MAXH2  10
#define MAXIT  10

#define FESTA 100

Real H(x,u,TDes,NEta)
Real *x,*u,*TDes;
SinSigno NEta;
{
   Real costo, *td;
   SinSigno i;
   costo=Tk;  /*Costo por defecto*/
   /*Las variables de estado tanto positivas como negativas ??*/
   /*X1 solo positiva mientras el resto positiva y negativa*/
   if(X1k<0)
      costo+=VARNEG*(-X1k);
 /*  else if(X1k>MAXH1)
      costo+=VARG*H1k;
   if(H2k<0)
      costo+=VARNEG*(-H2k);
   else if(H2k>MAXH2)
      costo+=VARG*H2k;
   if(ITk<0)
      costo+=VARNEG*(-ITk);
   else if(ITk>MAXH2)
      costo+=VARG*ITk;  */
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
      if(!isinf(td[i]))
        costo+=FESTA*SQR(x[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(x,u,TDes,NEta,re)
Real *x,*u,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;

   re[0]=0.0; /*Valors por defecto*/
   re[1]=0.0;
   re[2]=0.0;
   if(X1k<0)        /*Caso de salirse del rango*/
      re[0]=-VARNEG;
 /*else if(H1k>MAXH1)
      re[0]=VARG;
   if(H2k<0)
      re[1]=-VARNEG;
   else if(H2k>MAXH2)
      re[1]=VARG;
   if(ITk<0)
      re[2]=-VARNEG;
   else if(ITk>MAXH2)
     re[]VARG;*/
   if (NEta &&  TDes)
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      re[i]+=isinf(td[i])?0:FESTA*2*(x[i]-td[i]);
   }
}

void ParcialUdeH( x,u,re )
Real *x,*u,*re;
{
   re[0]=0.0; re[1]=0.0; re[2]=1.0;  return;
}

Real HN( Xn ) /*Dejamos libre X3N*/
Real *Xn;
{
   return Vn*(SQR(X2D-X2N)+SQR(X1D-X1N));
}

void ParcialXnHn( Xn,Ln )
Real *Xn,*Ln;
{
   Ln[0]=-2*Vn*(X1D-X1N);
   Ln[1]=-2*Vn*(X2D-X2N);
   Ln[2]=0;
   return;
}
```

A.2.3.2 p2l3l.h

```
/*Ecuaciones para el segundo metodo con discretzacion adelante
  el sisteme hilare con la suma de la cte en la segunda ecua.*/
  /*Solo se permite U1 positivo pero se libera X3N*/


/* ********** Constantes del problema ***************** */

#define X1D     0.0     /*Valores del punto final*/
#define X2D     0.0
#define X3D     0.0
#define A       0.001   /*Valor peque~nio del parametro*/

#define SQR(x) (x)*(x)

#define VAL_P 1e-30

/* ********** Funciones del problema ***************** */
#define X1k     xk[0]
#define X2k     xk[1]
#define X3k     xk[2]
#define X1k1     xk1[0]
#define X2k1     xk1[1]
#define X3k1     xk1[2]
#define U1k     uk[0]
#define U2k     uk[1]
#define Tk      uk[2]
#define U1k1     uk1[0]
#define U2k1     uk1[1]
#define Tk1      uk1[2]
#define dU1k1     re[0]
#define dU2k1     re[1]
#define dTk1      re[2]
#define X1N     Xn[0]
#define X2N     Xn[1]
#define X3N     Xn[2]


void F(xk,xk1,uk,uk1)
Real *xk,*xk1,*uk,*uk1;
{
   Tk1=((X2k1-X2k)-(X1k1-X1k)*tan(X3k))/A;
   U1k1=(X1k1-X1k)/(Tk1*cos(X3k));
   U2k1=(X3k1-X3k)/Tk1;
   return;
}

void ParcialXqF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   if(q==1)  /*respecto a X1k*/
   {
      dTk1=tan(X3k)/A;
      dU1k1=(-Tk1-(X1k1-X1k)*dTk1)/(cos(X3k)*Tk1*Tk1);
      dU2k1=(-(X3k1-X3k)*dTk1)/(Tk1*Tk1);
      return;
   }
   else if(q==2)  /*Respecto a X2k */
   {
      dTk1=-1/A;
      dU1k1=(-(X1k1-X1k)*dTk1)/(cos(X3k)*Tk1*Tk1);
      dU2k1=(-(X3k1-X3k)*dTk1)/(Tk1*Tk1);

      return;
   }
   else if (q==3)  /*respecto a X3k*/
   {
      dTk1=-(X1k1-X1k)/(A*cos(X3k)*cos(X3k));
      dU1k1=-(X1k1-X1k)*(dTk1*cos(X3k)-Tk1*sin(X3k))/SQR(Tk1*cos(X3k));
      dU2k1=(-Tk1-(X3k1-X3k)*dTk1)/(Tk1*Tk1);
      return;
   }
   else exit(3);  /* acceso incorrecto */
}


void ParcialX1qF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   if(q==1)  /*respecto a X1k1*/
   {
      dTk1=-tan(X3k)/A;
      dU1k1=(Tk1-(X1k1-X1k)*dTk1)/(cos(X3k)*Tk1*Tk1);
      dU2k1=-(X3k1-X3k)*dTk1/(Tk1*Tk1);
      return;
   }
   else if(q==2)  /*Respecto a X2k1 */
   {
      dTk1=1/A;
      dU1k1=-(X1k1-X1k)*dTk1/(cos(X3k)*Tk1*Tk1);
      dU2k1=-(X3k1-X3k)*dTk1/(Tk1*Tk1);
      return;
   }
   else if (q==3)  /*respecto a ITk1*/
   {
      dTk1=0;
      dU1k1=0;
      dU2k1=1/Tk1;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   if(q==1)  /*Respecto a U1k*/
   {
      dTk1=0.0; dU1k1=0.0; dU2k1=0.0; return;
   }
```

```
   else if(q==2) /*Respecto a U2k*/
   {
      dTk1=0.0; dU1k1=0.0; dU2k1=0.0; return;
   }
   else if(q==3) /*Respecto a Tk*/
   {
      dTk1=0.0; dU1k1=0.0; dU2k1=0.0; return;
   }
   else exit(3);  /* acceso incorrecto */
}


/*Obtamos por funcion de coste derivable en todos los puntos*/

/*Funcion de coste que garantiza tiempos positivos ya que da costo
   mayor para los tiempos negativos en todas las situaciones, incluso
   para -1<t<0 (13/4/93)
   -Pesamos tambien el comando para evitar que salga de cierto rango
     los caudales los deseamos solo positivos (21/4/93)*/
/*(28/6/93) Como ahora garantizamos que los pesos iniciales no dan
   trayectorias con tiempos negativos podemos pesar estos mucho mas
   y tomar para los tiempos positivos una funcion lineal*/

#define FTPOS 1       /*Igual que para el primer metodo*/
#define FTNEG  10000
#define RANU1 10       /*Ahora u1 solo positivo*/
#define RANU2 5
#define  FACG 100
#define Vn  10000    /*Igual que para el primer metodo*/
#define FESTA 100

Real H(xk,uk,TDes,NEta)
Real *xk,*uk,*TDes;
SinSigno NEta;
{
   Real costo, *td;
   SinSigno i;
   /*Costo por defecto*/
   costo=(Tk>=0)?FTPOS*Tk:1e6-Tk*FTNEG; /*No derivable*/
   if(U1k>RANU1)
      costo+=FACG*(U1k-RANU1);
   else if(U1k<0)
      costo+=FACG*(-U1k);
   if(U2k>RANU2)
      costo+=FACG*(U2k-RANU2);
   else if(U2k<-RANU2)
      costo+=FACG*(-RANU2-U2k);
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
  if(!isinf(td[i]))
            costo+=FESTA*(xk[i]-td[i])*(xk[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(xk,uk,TDes,NEta,re)
Real *xk,*uk,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;
   if (!NEta || !TDes)
   {
      re[0]=0.0; re[1]=0.0; re[2]=0.0; return;
   }
   else
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
         re[i]=isinf(td[i])?0:FESTA*2*(xk[i]-td[i]);
   }
}


void ParcialUdeH( xk,uk,re )
Real xk[VE],uk[VC],re[VC];
{
   dTk1=(Tk>=0)?FTPOS:-FTNEG;
   if(U1k>RANU1)
      dU1k1=FACG;
   else if(U1k<0)
      dU1k1=-FACG;
   else
      dU1k1=0.0;
   if(U2k>RANU2)
      dU2k1=FACG;
   else if(U2k<-RANU2)
      dU2k1=-FACG;
   else
      dU2k1=0.0;
   return;
}


Real HN( Xn )
Real *Xn;
{
   return Vn*(SQR(X2D-X2N)+SQR(X1D-X1N));
}

void ParcialXnHn( Xn,Ln )
Real *Xn,*Ln;
{
   Ln[0]=-2*Vn*(X1D-X1N);
   Ln[1]=-2*Vn*(X2D-X2N);
   Ln[2]=0;
   return;
}
```

A.2.3.3 p2bpf3l.h

```
/*Ecuaciones para el segundo metodo con discretzacion adelante
  el sisteme hilare con la suma de la cte en la segunda ecua.*/
  /*Solo permitido U1>0 pero liberando X3N */


/* ********** Constantes del problema ********** */

#define X1D     0.0     /*Valores del punto final*/
#define X2D     0.0
#define X3D     0.0
#define A       0.001   /*Valor peque~nio del parametro*/

#define SQR(x) (x)*(x)

#define VAL_P 1e-30

/* ********** Funciones del problema ********** */
#define X1k     xk[0]
#define X2k     xk[1]
#define X3k     xk[2]
#define X1k1     xk1[0]
#define X2k1     xk1[1]
#define X3k1     xk1[2]
#define U1k     uk[0]
#define U2k     uk[1]
#define Tk      uk[2]
#define U1k1     uk1[0]
#define U2k1     uk1[1]
#define Tk1      uk1[2]
#define dU1k1     re[0]
#define dU2k1     re[1]
#define dTk1      re[2]
#define X1N     Xn[0]
#define X2N     Xn[1]
#define X3N     Xn[2]


#define DENT (2*A+(sin(X3k)-tan(X3k1)*cos(X3k))*U1k)
void F(xk,xk1,uk,uk1)
Real *xk,*xk1,*uk,*uk1;
{
   Tk1=2*((X2k1-X2k)-(X1k1-X1k)*tan(X3k1))/DENT;
   U1k1=(2*(X1k1-X1k)/Tk1-cos(X3k)*U1k)/cos(X3k1);
   U2k1=2*(X3k1-X3k)/Tk1-U2k;
   return;
}

void ParcialXqF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   if(q==1)  /*respecto a X1k*/
   {
      dTk1=2*tan(X3k1)/DENT;
      dU1k1=2*(-Tk1-(X1k1-X1k)*dTk1)/(cos(X3k1)*Tk1*Tk1);
      dU2k1=-2*((X3k1-X3k)*dTk1)/(Tk1*Tk1);
      return;
   }
   else if(q==2)  /*Respecto a X2k */
   {
      dTk1=-2/DENT;
      dU1k1=(-2*(X1k1-X1k)*dTk1)/(cos(X3k1)*Tk1*Tk1);
      dU2k1=-2*((X3k1-X3k)*dTk1)/(Tk1*Tk1);

      return;
   }
   else if (q==3)  /*respecto a X3k*/
   {
      dTk1=-Tk1*U1k*(cos(X3k)+tan(X3k1)*sin(X3k))/DENT;
      dU1k1=(-2*(X1k1-X1k)*dTk1/(Tk1*Tk1)+sin(X3k)*U1k)/cos(X3k1);
      dU2k1=2*(-Tk1-(X3k1-X3k)*dTk1)/(Tk1*Tk1);
      return;
   }
   else exit(3);  /* acceso incorrecto */
}


void ParcialX1qF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   if(q==1)  /*respecto a X1k1*/
   {
      dTk1=-2*tan(X3k1)/DENT;
      dU1k1=2*(Tk1-(X1k1-X1k)*dTk1)/(cos(X3k1)*Tk1*Tk1);
      dU2k1=-2*(X3k1-X3k)*dTk1/(Tk1*Tk1);
      return;
   }
   else if(q==2)  /*Respecto a X2k1 */
   {
      dTk1=2/DENT;
      dU1k1=-2*(X1k1-X1k)*dTk1/(cos(X3k)*Tk1*Tk1);
      dU2k1=-2*(X3k1-X3k)*dTk1/(Tk1*Tk1);
      return;
   }
   else if (q==3)  /*respecto a ITk1*/
   {
      dTk1=(-2*(X1k1-X1k)+Tk1*cos(X3k)*U1k)/(cos(X3k1)*cos(X3k1)*DENT);
      dU1k1=(-2*(X1k1-X1k)*dTk1/(Tk1*Tk1)+U1k1*sin(X3k1))/cos(X3k1);
      dU2k1=2*(Tk1-(X3k1-X3k)*dTk1)/(Tk1*Tk1);
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   if(q==1)  /*Respecto a U1k*/
   {
```

```
      dTk1=-Tk1*(sin(X3k)-tan(X3k1)*cos(X3k))/DENT;
      dU1k1=(-2*(X1k1-X1k)*dTk1/(Tk1*Tk1)-cos(X3k))/cos(X3k1);
      dU2k1=-2*(X3k1-X3k)*dTk1/(Tk1*Tk1); return;
   }
   else if(q==2) /*Respecto a U2k*/
   {
      dTk1=0.0; dU1k1=0.0; dU2k1=-1; return;
   }
   else if(q==3) /*Respecto a Tk*/
   {
      dTk1=0.0; dU1k1=0.0; dU2k1=0.0; return;
   }
   else exit(3);  /* acceso incorrecto */
}

/*Obtamos por funcion de coste derivable en todos los puntos*/

/*Funcion de coste que garantiza tiempos positivos ya que da costo
   mayor para los tiempos negativos en todas las situaciones, incluso
   para -1<t<0 (13/4/93)
   -Pesamos tambien el comando para evitar que salga de cierto rango
     los caudales los deseamos solo positivos (21/4/93)*/
/*(28/6/93) Como ahora garantizamos que los pesos iniciales no dan
   trayectorias con tiempos negativos podemos pesar estos mucho mas
   y tomar para los tiempos positivos una funcion lineal*/

#define FTPOS 1       /*Igual que para el primer metodo*/
#define FTNEG  10000
#define RANU1 10      /*Ahora solo positivo*/
#define RANU2 5
#define  FACG 100
#define Vn  10000    /*Igual que para el primer metodo*/
#define FESTA 100

Real H(xk,uk,TDes,NEta)
Real *xk,*uk,*TDes;
SinSigno NEta;
{
   Real costo, *td;
   SinSigno i;
   /*Costo por defecto*/
   costo=(Tk>=0)?FTPOS*Tk:1e6-Tk*FTNEG; /*No derivable*/
   if(U1k>RANU1)
      costo+=FACG*(U1k-RANU1);
   else if(U1k<0)
      costo+=FACG*(-U1k);
   if(U2k>RANU2)
      costo+=FACG*(U2k-RANU2);
   else if(U2k<-RANU2)
      costo+=FACG*(-RANU2-U2k);
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
  if(!isinf(td[i]))
         costo+=FESTA*(xk[i]-td[i])*(xk[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(xk,uk,TDes,NEta,re)
Real *xk,*uk,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;
   if (!NEta || !TDes)
   {
      re[0]=0.0; re[1]=0.0; re[2]=0.0; return;
   }
   else
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
         re[i]=isinf(td[i])?0:FESTA*2*(xk[i]-td[i]);
   }
}


void ParcialUdeH( xk,uk,re )
Real xk[VE],uk[VC],re[VC];
{
   dTk1=(Tk>=0)?FTPOS:-FTNEG;
   if(U1k>RANU1)
      dU1k1=FACG;
   else if(U1k<0)
      dU1k1=-FACG;
   else
      dU1k1=0.0;
   if(U2k>RANU2)
      dU2k1=FACG;
   else if(U2k<-RANU2)
      dU2k1=-FACG;
   else
      dU2k1=0.0;
   return;
}


Real HN( Xn )
Real *Xn;
{
   return Vn*(SQR(X2D-X2N)+SQR(X1D-X1N));
}

void ParcialXnHn( Xn,Ln )
Real *Xn,*Ln;
{
   Ln[0]=-2*Vn*(X1D-X1N);
   Ln[1]=-2*Vn*(X2D-X2N);
   Ln[2]=0;
   return;
}
```

A.2.4 Tercer Sistema

A.2.4.1 pro1pd.h

```
/*Sistema mimo metodo1 lineal ctes. de PD*/

/* ********** Constantes del problema *************** */
#define H0  5   /*diferencia de alturas entre los depositos*/
#define R1  2.8
#define R2  R1
#define RT  1     /*resistencia termica del aislamiento*/
#define CS  1.5  /*calor especifico del liquido + Ce aluminio*/
#define CP  1    /*calor especifico del liquido*/

#define H2D  5 /*altura deseada 5 cm*/
#define H1D  ((1+R2/R1)*H2D-H0)
#define ITD  5  /*diferencia de temperatura deseada*/

#define P(x)  (x)
#define DerP(x)   (1)
#define SQR(x) (x)*(x)


/* ********** Funciones del problema ***************** */
#define H1k     x[0]
#define H2k     x[1]
#define ITk     x[2]
#define QLk     u[0]
#define QCk     u[1]
#define Tk      u[2]
#define H1k1    re[0]
#define H2k1    re[1]
#define ITk1    re[2]
#define H1N     Xn[0]
#define H2N     Xn[1]
#define ITN     Xn[2]

#define DHk     (H1k-H2k+H0)

void F(x,u,re)
Real *x,*u,*re;
{
   H1k1=H1k+Tk*(-R1*P(DHk)+QLk);
   H2k1=H2k+Tk*(R1*P(DHk)-R2*P(H2k));
   ITk1=ITk+Tk*((-1/RT-CP*(R1*P(DHk)-R2*P(H2k)))*ITk+QCk)/CS;
   return;
}

void ParcialXqF(x,u,q,re)
Real *x,*u,*re;
SinSigno q;
{
   if(q==1)  /*respecto a H1k*/
   {
      H1k1=1-Tk*R1*DerP(DHk);
      H2k1=Tk*R1*DerP(DHk);
      ITk1=-Tk*ITk*CP*R1*DerP(DHk)/CS;
      return;
   }
   else if(q==2)  /*Respecto a H2k */
   {
      H1k1=Tk*R1*DerP(DHk);
      H2k1=1+Tk*(-R1*DerP(DHk)-R2*DerP(H2k));
      ITk1=-Tk*CP*ITk*(-R1*DerP(DHk)-R2*DerP(H2k))/CS;
      return;
   }
   else if (q==3)  /*respecto a ITk*/
   {
      H1k1=0; H2k1=0;
      ITk1=1+Tk*(-1/RT-CP*(R1*P(DHk)-R2*P(H2k)))/CS;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(x,u,q,re)
Real *x,*u,*re;
SinSigno q;
{
   if(q==1)  /*Respecto a QLk*/
   {
      H1k1=Tk; H2k1=0.0; ITk1=0.0; return;
   }
   else if(q==2) /*Respecto a QCk*/
   {
      H1k1=0.0; H2k1=0.0; ITk1=Tk/CS; return;
   }
   else if(q==3) /*Respecto a Tk*/
   {
      H1k1=-R1*P(DHk)+QLk;
      H2k1=R1*P(DHk)-R2*P(H2k);
      ITk1=((-1/RT-CP*(R1*P(DHk)-R2*P(H2k)))*ITk+QCk)/CS;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

#define Vn  10000
#define VARNEG 100   /*Pendiente var. est. negativas*/
#define VARG   100   /*Pendiente var. est. grandes*/
#define MAXH1  10
#define MAXH2  10
#define MAXIT  10

#define FESTA 100

Real H(x,u,TDes,NEta)
Real *x,*u,*TDes;
SinSigno NEta;
{
   Real costo, *td;
```

```
   SinSigno i;
   costo=Tk;  /*Costo por defecto*/
   if(H1k<0)
      costo+=VARNEG*(-H1k);
   else if(H1k>MAXH1)
      costo+=VARG*H1k;
   if(H2k<0)
      costo+=VARNEG*(-H2k);
   else if(H2k>MAXH2)
      costo+=VARG*H2k;
   if(ITk<0)
      costo+=VARNEG*(-ITk);
   else if(ITk>MAXH2)
      costo+=VARG*ITk;
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
      if(!isinf(td[i]))
        costo+=FESTA*SQR(x[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(x,u,TDes,NEta,re)
Real *x,*u,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;

   re[0]=0.0; /*Valors por defecto*/
   re[1]=0.0;
   re[2]=0.0;
   if(H1k<0)       /*Caso de salirse del rango*/
      re[0]=-VARNEG;
   else if(H1k>MAXH1)
      re[0]=VARG;
   if(H2k<0)
      re[1]=-VARNEG;
   else if(H2k>MAXH2)
      re[1]=VARG;
   if(ITk<0)
      re[2]=-VARNEG;
   else if(ITk>MAXH2)
      re[2]=VARG;
   if (NEta &&  TDes)
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      re[i]+=isinf(td[i])?0:FESTA*2*(x[i]-td[i]);
   }
}

void ParcialUdeH( x,u,re )
Real *x,*u,*re;
{
   re[0]=0.0; re[1]=0.0; re[2]=1.0;  return;
}

Real HN( Xn )
Real *Xn;
{
   return Vn*(SQR(H2D-H2N)+SQR(ITD-ITN)+SQR(H1D-H2N));
}

void ParcialXnHn( Xn,Ln )
Real *Xn,*Ln;
{
   Ln[0]=-2*Vn*(H1D-H1N);
   Ln[1]=-2*Vn*(H2D-H2N);
   Ln[2]=-2*Vn*(ITD-ITN);
   return;
}
```

A.2.4.2 p2pdcl2.h

```
/* funciones para la ecuacion del sistema MIMO discretizada
   con differencia alante Para el metodo 2*/
/*Modificada 15/2/95 para usar ctes. y pto final de la PD*/

/* ********** Constantes del problema ********** */
#define H0  5   /*diferencia de alturas entre los depositos*/
#define R1  2.8
#define R2  R1
#define RT  1     /*resistencia termica del aislamiento*/
#define CS  1.5  /*calor especifico del liquido + Ce aluminio*/
#define CP  1    /*calor especifico del liquido*/

#define H2D  5    /*altura deseada 5 cm*/
#define H1D  ((1+R2/R1)*H2D-H0)  /*h1 para punto estable*/
#define ITD  5  /*diferencia de temperatura deseada*/

#define P(x)  (x)
#define DerP(x)  1
#define SQR(x) (x)*(x)

#define VAL_P 1e-30

/* ********** Funciones del problema ********** */
#define H1k     xk[0]
#define H2k     xk[1]
#define ITk     xk[2]
#define H1k1    xk1[0]
#define H2k1    xk1[1]
#define ITk1    xk1[2]
#define QLk     uk[0]
#define QCk     uk[1]
#define Tk      uk[2]
#define QLk1    uk1[0]
#define QCk1    uk1[1]
#define Tk1     uk1[2]
```

```
#define dQLk1    re[0]
#define dQCk1    re[1]
#define dTk1     re[2]
#define H1N      Xn[0]
#define H2N      Xn[1]
#define ITN      Xn[2]

#define DHk      (H1k-H2k+H0)
#define DHk1     (H1k1-H2k1+H0)

void F(xk,xk1,uk,uk1)
Real *xk,*xk1,*uk,*uk1;
{
   Real dhk;
   dhk=DHk;

   Tk1=(H2k1-H2k)/(R1*P(dhk)-R2*P(H2k));
   if (!Tk1)
     Tk1=VAL_P;
   QLk1=(H1kI-H1k)/Tk1+R1*P(dhk);
   QCk1=CS*(ITk1-ITk)/Tk1+ITk*(1/RT+CP*(R1*P(dhk)-R2*P(H2k)));
   return;
}

void ParcialXqF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   Real dhk;
   dhk=DHk;

   if(q==1)  /*respecto a H1k*/
   {
      dTk1=-Tk1*R1/(R1*dhk-R2*H2k);
      dQLk1=-(Tk1+(H1k1-H1k)*dTk1)/(Tk1*Tk1)+R1;
      dQCk1=-CS*(ITk1-ITk)*dTk1/(Tk1*Tk1)+ITk*CP*R1;
      return;
   }
   else if(q==2)  /*Respecto a H2k */
   {
      dTk1=(-1+Tk1*(R1+R2))/(R1*P(dhk)-R2*P(H2k));
      dQLk1=-(H1k1-H1k)/(Tk1*Tk1)*dTk1-R1;
      dQCk1=-CS*(ITk1-ITk)/(Tk1*Tk1)*dTk1-ITk*CP*(R1+R2);
      return;
   }
   else if (q==3)  /*respecto a ITk*/
   {
      dTk1=0;
      dQLk1=0;
      dQCk1=-CS/Tk1+(1/RT+CP*(R1*dhk-R2*H2k));
      return;
   }
   else exit(3);  /* acceso incorrecto */
}


void ParcialX1qF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   Real dhk;
   dhk=DHk;

   if(q==1)  /*respecto a H1k1*/
   {
      dTk1=0;
      dQLk1=1/Tk1;
      dQCk1=0;
      return;
   }
   else if(q==2)  /*Respecto a H2k1 */
   {
      dTk1=1/(R1*dhk-R2*H2k);
      dQLk1=-(H1k1-H1k)*dTk1/(Tk1*Tk1);
      dQCk1=-CS*(ITk1-ITk)/(Tk1*Tk1)*dTk1;
      return;
   }
   else if (q==3)  /*respecto a ITk1*/
   {
      dTk1=0;
      dQLk1=0;
      dQCk1=CS/Tk1;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   if(q==1)  /*Respecto a QLk*/
   {
      dTk1=0.0; dQLk1=0.0; dQCk1=0.0; return;
   }
   else if(q==2) /*Respecto a QCk*/
   {
      dTk1=0.0; dQLk1=0.0; dQCk1=0.0; return;
   }
   else if(q==3) /*Respecto a Tk*/
   {
      dTk1=0.0; dQLk1=0.0; dQCk1=0.0; return;
   }
   else exit(3);  /* acceso incorrecto */
}


/*Obtamos por funcion de coste derivable en todos los puntos*/

/*Funcion de coste que garantiza tiempos positivos ya que da costo
   mayor para los tiempos negativos en todas las situaciones, incluso
   para -1<t<0 (13/4/93)
   -Pesamos tambien el comando para evitar que salga de cierto rango
      los caudales los deseamos solo positivos (21/4/93)*/
/*(28/6/93) Como ahora garantizamos que los pesos iniciales no dan
   trayectorias con tiempos negativos podemos pesar estos mucho mas
```

```
   y tomar para los tiempos positivos una funcion lineal*/

/*#define FTPOS 1000  1/6/95 cambiamos al valor del 1er. metodo*/
#define FTPOS 1
#define FTNEG  10000
#define RANL 50
#define RANC 50
/*#define  FACG 50 1/6/95 para eliminar los comandos negativos*/
#define  FACG 500
#define Vn  100
#define FESTA 100


Real H(xk,uk,TDes,NEta)
Real *xk,*uk,*TDes;
SinSigno NEta;
{
   Real costo, *td;
   SinSigno i;
   /*Costo por defecto*/
   costo=(Tk>=0)?FTPOS*Tk:1000000-Tk*FTNEG; /*No derivable*/
   if(QLk>RANL)
      costo+=FACG*(QLk-RANL);
   else if(QLk<0)
      costo+=-FACG*QLk;
   if(QCk>RANC)
      costo+=FACG*(QCk-RANC);
   else if(QCk<0)
      costo+=-FACG*QCk;
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
  if(!isinf(td[i]))
         costo+=FESTA*(xk[i]-td[i])*(xk[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(xk,uk,TDes,NEta,re)
Real *xk,*uk,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;
   if (!NEta || !TDes)
   {
      re[0]=0.0; re[1]=0.0; re[2]=0.0; return;
   }
   else
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
         re[i]=isinf(td[i])?0:FESTA*2*(xk[i]-td[i]);
   }
}


void ParcialUdeH( xk,uk,re )
Real xk[VE],uk[VC],re[VC];
{
   dTk1=(Tk>=0)?FTPOS:-FTNEG;
   if(QLk>RANL)
      dQLk1=FACG;
   else if(QLk<0)
      dQLk1=-FACG;
   else
      dQLk1=0.0;
   if(QCk>RANC)
      dQCk1=FACG;
   else if(QCk<0)
      dQCk1=-FACG;
   else
      dQCk1=0.0;
   return;
}


Real HN( Xn )
Real *Xn;
{
   return Vn*(SQR(H2D-H2N)+SQR(ITD-ITN)+SQR(H1D-H1N));
}

void ParcialXnHn( Xn,Ln )
Real *Xn,*Ln;
{
   Ln[0]=-2*Vn*(H1D-H1N);
   Ln[1]=-2*Vn*(H2D-H2N);
   Ln[2]=-2*Vn*(ITD-ITN);
   return;
}
```

A.2.4.3 p2pdbp.h

```
/* funciones para la ecuacion de Van der Pol discretizada
   con BPF Para el metodo 2*/
/*Modificada 15/2/95 para usar ctes. y pto final de la PD*/

/* ********** Constantes del problema *************** */
#define H0  5   /*diferencia de alturas entre los depositos*/
#define R1  2.8
#define R2  R1
#define RT  1     /*resistencia termica del aislamiento*/
#define CS  1.5  /*calor especifico del liquido + Ce aluminio*/
#define CP  1    /*calor especifico del liquido*/

#define H2D  5    /*altura deseada 5 cm*/
#define H1D  ((1+R2/R1)*H2D-H0)  /*h1 para punto estable*/
#define ITD  5  /*diferencia de temperatura deseada*/

#define P(x)  (x)
```

```
#define DerP(x)  1
#define SQR(x) (x)*(x)

#define VAL_P 1e-30

/* ********** Funciones del problema ***************** */
#define H1k      xk[0]
#define H2k      xk[1]
#define ITk      xk[2]
#define H1k1     xk1[0]
#define H2k1     xk1[1]
#define ITk1     xk1[2]
#define QLk      uk[0]
#define QCk      uk[1]
#define Tk       uk[2]
#define QLk1     uk1[0]
#define QCk1     uk1[1]
#define Tk1      uk1[2]
#define dQLk1    re[0]
#define dQCk1    re[1]
#define dTk1     re[2]
#define H1N      Xn[0]
#define H2N      Xn[1]
#define ITN      Xn[2]

#define DHk      (H1k-H2k+H0)
#define DHk1     (H1k1-H2k1+H0)

void F(xk,xk1,uk,uk1)
Real *xk,*xk1,*uk,*uk1;
{
   Real dhk1,dhk;
   dhk1=DHk1;
   dhk=DHk;

   Tk1=(2*(H2k1-H2k)+Tk*(-R1*dhk1+R2*H2k))/(R1*dhk1-R2*H2k1);
   if (!Tk1)
     Tk1=VAL_P;
   QLk1=(2*(H1k1-H1k)+R1*(Tk*dhk+Tk1*dhk1)-Tk*QLk)/Tk1;
   QCk1=(2*CS*(ITk1-ITk)+Tk*(ITk*(1/RT+CP*(R1*dhk-R2*H2k))+QCk)+
       Tk1*ITk1*(1/RT+CP*(R1*dhk1-R2*H2k1)))/Tk1/CS;
   return;
}

void ParcialXqF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   Real dhk1,dhk;
   dhk1=DHk1;
   dhk=DHk;

   if(q==1)  /*respecto a H1k*/
   {
      dTk1=(-R1*Tk)/(R1*dhk1-R2*H2k1);
      dQLk1=(-2+R1*Tk-(QLk1-R1*dhk1)*dTk1)/Tk1;
      dQCk1=(CP*Tk*ITk*R1-(QCk1*CS-ITk1/RT-CP*ITk1*(R1*dhk1-R2*H2k1))
  *dTk1)/Tk1/CS;
      return;
   }
   else if(q==2)  /*Respecto a H2k */
   {
      dTk1=(-2+R1*Tk+R2*Tk)/(R1*dhk1-R2*H2k1);
      dQLk1=(-R1*Tk-(QLk1-R1*dhk1)*dTk1)/Tk1;
      dQCk1=(CP*Tk*ITk*(-R1-R2)-(CS*QCk1-ITk1/RT-CP*ITk1*
  (R1*dhk1-R2*H2k1))*dTk1)/(Tk1*CS);
      return;
   }
   else if (q==3)  /*respecto a ITk*/
   {
      dTk1=0;
      dQLk1=0;
      dQCk1=(-2*CS+Tk/RT+CP*Tk*(R1*dhk-R2*H2k))/(Tk1*CS);
      return;
   }
   else exit(3);  /* acceso incorrecto */
}


void ParcialX1qF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
   Real dhk1;
   dhk1=DHk1;

   if(q==1)  /*respecto a H1k1*/
   {
      dTk1=-Tk1*R1/(R1*dhk1-R2*H2k1);
      dQLk1=(2-(QLk1-R1*dhk1)*dTk1)/Tk1+R1;
      dQCk1=-(CS*QCk1-ITk1/RT-CP*ITk1*(R1*dhk1-R2*H2k1))*dTk1/(Tk1*CS)+
  CP*ITk1*R1/CS;
      return;
   }
   else if(q==2)  /*Respecto a H2k1 */
   {
      dTk1=(2+Tk1*(R1+R2))/(R1*dhk1-R2*H2k1);
      dQLk1=-(QLk1-R1*dhk1)*dTk1/Tk1-R1;
      dQCk1=-(CS*QLk1-ITk1/RT-CP*ITk1*(R1*dhk1-R2*H2k1))*dTk1/(Tk1*CS)-
 CP*ITk1*(R1+R2)/CS;
      return;
   }
   else if (q==3)  /*respecto a ITk1*/
   {
      dTk1=0;
      dQLk1=0;
      dQCk1=2/Tk1+1/CS/RT+CP*(R1*dhk1-R2*H2k1)/CS;
      return;
   }
   else exit(3);  /* acceso incorrecto */
}

void ParcialUqF(xk,xk1,uk,uk1,q,re)
Real *xk,*xk1,*uk,*uk1,*re;
SinSigno q;
{
```

```
   Real dhk1,dhk;
   dhk1=DHk1;
   dhk=DHk;

   if(q==1)  /*Respecto a QLk*/
   {
      dTk1=0.0; dQLk1=-Tk/Tk1; dQCk1=0.0; return;
   }
   else if(q==2) /*Respecto a QCk*/
   {
      dTk1=0.0; dQLk1=0.0; dQCk1=-Tk/Tk1; return;
   }
   else if(q==3) /*Respecto a Tk*/
   {
      dTk1=(-R1*dhk+R2*H2k)/(R1*dhk1-R2*H2k1);
      dQLk1=(R1*dhk-QLk-(QLk1-R1*dhk1)*dTk1)/Tk1;
      dQCk1=((ITk/RT+CP*ITk*(R1*dhk-R2*H2k)-CS*QCk)-
  (CS*QCk1-ITk1/RT-ITk1*CP*(R1*dhk1-R2*H2k1))*dTk1)/(Tk1*CS);
      return;
   }
   else exit(3);  /* acceso incorrecto */
}


/*Obtamos por funcion de coste derivable en todos los puntos*/

/*Funcion de coste que garantiza tiempos positivos ya que da costo
   mayor para los tiempos negativos en todas las situaciones, incluso
   para -1<t<0 (13/4/93)
   -Pesamos tambien el comando para evitar que salga de cierto rango
     los caudales los deseamos solo positivos (21/4/93)*/
/*(28/6/93) Como ahora garantizamos que los pesos iniciales no dan
   trayectorias con tiempos negativos podemos pesar estos mucho mas
   y tomar para los tiempos positivos una funcion lineal*/

#define FTPOS 1000
#define FTNEG  10000
#define RANL 5
#define RANC 5
#define  FACG 50
#define Vn  100
#define FESTA 100

Real H(xk,uk,TDes,NEta)
Real *xk,*uk,*TDes;
SinSigno NEta;
{
   Real costo, *td;
   SinSigno i;
   /*Costo por defecto*/
   costo=(Tk>=0)?FTPOS*Tk:1000000-Tk*FTNEG; /*No derivable*/
   if(QLk>RANL)
      costo+=FACG*(QLk-RANL)*(QLk-RANL);
   else if(QLk<0)
      costo+=FACG*QLk*QLk;
   if(QCk>RANC)
      costo+=FACG*(QCk-RANC)*(QCk-RANC);
   else if(QCk<0)
      costo+=FACG*QCk*QCk;
   if(TDes && NEta>0)  /*Costo si existe trayectoria deseada*/
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
      /*Distancia al cuadrado*/
  if(!isinf(td[i]))
            costo+=FESTA*(xk[i]-td[i])*(xk[i]-td[i]);
   }
   return costo;
}

void ParcialXdeH(xk,uk,TDes,NEta,re)
Real *xk,*uk,*TDes,*re;
SinSigno NEta;
{
   Real *td;
   SinSigno i;
   if (!NEta || !TDes)
   {
      re[0]=0.0; re[1]=0.0; re[2]=0.0; return;
   }
   else
   {
      td=TDes+VE*(NEta-1);
      for(i=0; i<VE; i++)
         re[i]=isinf(td[i])?0:FESTA*2*(xk[i]-td[i]);
   }
}


void ParcialUdeH( xk,uk,re )
Real xk[VE],uk[VC],re[VC];
{
   dTk1=(Tk>=0)?FTPOS:-FTNEG;
   if(QLk>RANL)
      dQLk1=2*FACG*(QLk-RANL);
   else if(QLk<0)
      dQLk1=2*FACG*QLk;
   else
      dQLk1=0.0;
   if(QCk>RANC)
      dQCk1=2*FACG*(QCk-RANC);
   else if(QCk<0)
      dQCk1=2*FACG*QCk;
   else
      dQCk1=0.0;
   return;
}


Real HN( Xn )
Real *Xn;
{
   return Vn*(SQR(H2D-H2N)+SQR(ITD-ITN)+SQR(H1D-H2N));
}

void ParcialXnHn( Xn,Ln )
```

```
Real *Xn,*Ln;
{
   Ln[0]=-2*Vn*(H1D-H1N);
   Ln[1]=-2*Vn*(H2D-H2N);
   Ln[2]=-2*Vn*(ITD-ITN);
   return;
}
```

A.3 Tratamiento de Resultados

A.3.1 Manejo de Trayectorias

A.3.1.1 calPar.m

```
function MPar=calPar(tra,pes,npru,ran,Estru)
%MPar=calPar(tra,pes) calcula matriz de parametros con trajec. y pesos
% Formato:
%    MPar=calPar(tra,pes,npru,ran,Estru)
% tra,pes        Obligatorios
% resto          si existen se pasan a midsen
%Matriz de parametros:
% tantas comumnas como unidades
% las filas son:
%        Maximos Valores Singulares de cada capa
%        Minimos Valores Singulares de cada capa
%        Maximos Valores Singulares de cada capa sin Bias
%        Minimos Valores Singulares de cada capa sin Bias
%        Sensibilidad por capas
%        Sensibilidad completa

if(nargin<2)
   error('Funcion necesita al menos 2 parametros');
end
if (nargin>=5)
   [Mvs,mvs,MvsnoB,mvsnoB]=valsin(pes,Estru);
else
   [Mvs,mvs,MvsnoB,mvsnoB]=valsin(pes);
end
if(nargin>=3)
   if(nargin>=4)
      if(nargin>=5)
   senC=midsenC(tra,pes,npru,ran,Estru);
   senT=midsen(tra,pes,npru,ran,0,Estru);
      else
   senC=midsenC(tra,pes,npru,ran);
   senT=midsen(tra,pes,npru,ran);
      end
   else
      senC=midsenC(tra,pes,npru);
      senT=midsen(tra,pes,npru);
   end
else
   senC=midsenC(tra,pes);
   senT=midsen(tra,pes);
end

MPar=[Mvs;mvs;MvsnoB;mvsnoB;senC;senT];
```

A.3.1.2 caldt.m

```
% fichero M para obtener los datos de las matrices cargadas
%   evolucion de los distintos parametros.
%   matrices necesarias:
%      pre = de texto indica el prefijo (sin DT);
%      ind = rango de sufijo de ficheros a juntar
%   Matriz resultado 'DT' contendr- por columnas:
%      1-Numero iteraciones      2-Modulo del gradiente
%      3-Factor Eta              4-Modulo en la direcciãn de Busca
%      5-Alfa  encontrado        6-Fo  7-Fa  8-Fb
%      9-Maxima componente del gradiente
%      11-Maximo error Teorico/Practico del gradiente
%      12-Error medio      "       "       "       "
%  Se borran las matrices reunidas

if (isstr(pre))
   for i=ind
      fich=[pre,int2str(i)];
      if(exist([fich,'DT'])==1)
   eval(['DT=[DT;',fich,'DT];']);
   eval(['clear ',fich,'DT']);
      end
      if(exist([fich,'P'])==1)
         %Calculamos los parametros si estan los pesos
         eval([fich 'PAR=calPar(' fich  ',' fich 'P);']);
      end
   end
   if(exist([pre 'MEJ'])==1 & exist([pre 'MEJP'])==1)
      %Esta la mejor y sus pesos=> paramatros
      eval([pre 'MEJPAR=calPar(' pre 'MEJ,' pre 'MEJP);']);
   end
end
clear i fich ind pre
```

A.3.1.3 caldtm.m

```
% fichero M para obtener los datos de las matrices cargadas
%   evolucion de los distintos parametros.
%   matrices necesarias:
```

```
%      pre = de texto indica el prefijo (sin DT);
%      ind = rango de sufijo de ficheros a juntar
%    Matriz resultado 'DT' contendr- por columnas:
%      1-Numero iteraciones      2-Modulo del gradiente
%      3-Factor Eta              4-Modulo en la direcciãn de Busca
%      5-Alfa  encontrado        6-Fo  7-Fa  8-Fb
%      9-Maxima componente del gradiente
%      11-Maximo error Teorico/Practico del gradiente
%      12-Error medio      "       "       "       "
%  Se borran las matrices reunidas

if (isstr(pre))
   for i=ind
      fich=[pre,int2str(i)];
      if(exist([fich,'DT'])==1)
   eval(['DT=[DT;',fich,'DT];']);
   eval(['clear ',fich,'DT']);
      end
      if(exist([fich,'P'])==1)
         %Calculamos los parametros si estan los pesos
    eval([fich 'PAR=calPar(' fich  ',' fich 'P,100,1e-3,[3,5,3]);']);
      end
    end
    if(exist([pre 'MEJ'])==1 & exist([pre 'MEJP'])==1)
       %Esta la mejor y sus pesos=> paramatros
 eval([pre 'MEJPAR=calPar(' pre 'MEJ,' pre 'MEJP,100,1e-3,[3,5,3]);']);
    end
end
clear i fich ind pre
```

A.3.1.4 calg.m

```
% fichero M para obtener los datos de las matrices cargadas
%   evolucion de los distintos parametros.
%   matrices necesarias:
%      pre = de texto indica el prefijo (sin DT);
%      ind = rango de sufijo de ficheros a juntar
%   Matriz resultado 'DT' contendr- por columnas:
%      1-Numero iteraciones      2-Modulo del gradiente
%      3-Factor Eta              4-Modulo en la direcciãn de Busca
%      5-Alfa  encontrado        6-Fo  7-Fa  8-Fb
%      9-Maxima componente del gradiente
%      11-Maximo error Teorico/Practico del gradiente
%      12-Error medio      "       "       "       "
%  Se borran las matrices reunidas

if (isstr(pre))
   eval(['load ',pre,int2str(ind(length(ind)))]);
   for i=ind
      fich=[pre,int2str(i)];
      if(exist([fich,'DT'])==1)
   eval(['DT=[DT;',fich,'DT ];']);
   eval(['clear ',fich,'DT']);
      end
      if(exist([fich,'P'])==1)
   %Calculamos los parametros si estan los pesos
   eval([fich,'PAR=calPar(' fich ',' fich 'P);']);
      end
    end
    if(exist([pre 'MEJ'])==1 & exist([pre 'MEJP'])==1)
       %Esta la mejor y sus pesos=> paramatros
       eval([pre 'MEJPAR=calPar(' pre 'MEJ,' pre 'MEJP);']);
    end
end
clear i fich ind pre
```

A.3.1.5 calgm.m

```
% fichero M para obtener los datos de las matrices cargadas
%   evolucion de los distintos parametros.
%   matrices necesarias:
%      pre = de texto indica el prefijo (sin DT);
%      ind = rango de sufijo de ficheros a juntar
%   Matriz resultado 'DT' contendr- por columnas:
%      1-Numero iteraciones      2-Modulo del gradiente
%      3-Factor Eta              4-Modulo en la direcciãn de Busca
%      5-Alfa  encontrado        6-Fo  7-Fa  8-Fb
%      9-Maxima componente del gradiente
%      11-Maximo error Teorico/Practico del gradiente
%      12-Error medio      "       "       "       "
%  Se borran las matrices reunidas

if (isstr(pre))
   eval(['load ',pre,int2str(ind(length(ind)))]);
   for i=ind
      fich=[pre,int2str(i)];
      if(exist([fich,'DT'])==1)
   eval(['DT=[DT;',fich,'DT ];']);
   eval(['clear ',fich,'DT']);
      end
      if(exist([fich,'P'])==1)
   %Calculamos los parametros si estan los pesos
     eval([fich,'PAR=calPar(' fich ',' fich 'P,100,1e-3,[3,5,3]);']);
      end
    end
    if(exist([pre 'MEJ'])==1 & exist([pre 'MEJP'])==1)
       %Esta la mejor y sus pesos=> paramatros
 eval([pre 'MEJPAR=calPar(' pre 'MEJ,' pre 'MEJP,100,1e-3,[3,5,3]);']);
    end
end
clear i fich ind pre
```

A.3.1.6 calgm10.m

```
% fichero M para obtener los datos de las matrices cargadas
%   evolucion de los distintos parametros.
%   matrices necesarias:
```

```
%      pre = de texto indica el prefijo (sin DT);
%      ind = rango de sufijo de ficheros a juntar
%   Matriz resultado 'DT' contendr- por columnas:
%      1-Numero iteraciones      2-Modulo del gradiente
%      3-Factor Eta              4-Modulo en la direcciãn de Busca
%      5-Alfa  encontrado        6-Fo  7-Fa  8-Fb
%      9-Maxima componente del gradiente
%      11-Maximo error Teorico/Practico del gradiente
%      12-Error medio      "       "       "       "
%  Se borran las matrices reunidas

if (isstr(pre))
   eval(['load ',pre,int2str(ind(length(ind)))]);
   for i=ind
      fich=[pre,int2str(i)];
      if(exist([fich,'DT'])==1)
   eval(['DT=[DT;',fich,'DT ];']);
   eval(['clear ',fich,'DT']);
      end
      if(exist([fich,'P'])==1)
   %Calculamos los parametros si estan los pesos
    eval([fich,'PAR=calPar(' fich ',' fich 'P,100,1e-3,[3,10,3]);']);
      end
   end
   if(exist([pre 'MEJ'])==1 & exist([pre 'MEJP'])==1)
      %Esta la mejor y sus pesos=> paramatros
 eval([pre 'MEJPAR=calPar(' pre 'MEJ,' pre 'MEJP,100,1e-3,[3,10,3]);']);
   end
end
clear i fich ind pre
```

A.3.1.7 contiene.m

```
function r=contiene(t1,t2,t3,t4,t5,t6,t7,t8,t9)
% r=contiene(t1[,t2,..,t8]) puntos iniciales y costo de las trayect.
%  da punto inicial ,final y costo  de las
%  trayectorias que contiene cada una de las matrices t

trc=[];
%Coleccionamos los parametros de entrada
if (nargin>=1)
   if (length(t1)==1)
      for i=2:nargin
   t1
   trc=eval([ '[trc;sacan(t' int2str(i) ',t1)]' ]);
      end
   else
      for i=1:nargin
   trc=eval(['[trc;t',int2str(i),']']);
      end
   end
   r=[];
   [ft,ct]=size(trc);
   pt=4;
   while(pt<=ft)
      npt=trc(pt-3,1);

      if(ct==5)
          r=[ r
          trc(pt-2,1),trc(pt,1:2),trc(pt+npt-1,1:2),trc(pt+npt-1,5)];
      else if (ct==7)
   r=[r
     trc(pt-2,1),trc(pt,1:3),trc(pt+npt-1,1:3),trc(pt+npt-1,7)];
          end
      end
      pt=pt+npt+3;
   end
else
  echo "falta matriz trayectoria"
end
```

A.3.1.8 costo.m

```
function c=costo(t1,t2,t3,t4,t5,t6,t7,t8)
%c=costo(t1,...,t8) los costos cada por columnas e indice del menor

c=[];
for i=1:nargin
   eval(['conti=contiene(t' int2str(i) ');']);
   c=[c conti(:,size(conti,2))];
end
[m,ind]=min(c');
c=[c ind'];
```

A.3.1.9 costomi.m

```
function costomi(tra1,tra2,tra3,tra4)
% pinta costo de trayectorias
% Permite hasta 4 trayectorias
%    function costomi(tra1,tra2,tra3,tra4)
%  trai -> trayectoria con cabecera, si existen varias, la primera

tra=['r';'g';'y';'m'];

t1=[]; t2=[]; t3=[]; t4=[];
l1=1;  l2=1;  l3=1;  l4=1;
if (nargin>=1)
   t1=sacatn(tra1);
   [l1,an1]=size(t1);
   if (nargin>=2)
      t2=sacatn(tra2);
      [l2,an1]=size(t2);
      if (nargin>=3)
   t3=sacatn(tra3);
   [l3,an3]=size(t3);
```

```
   if (nargin>=4)
      t4=sacatn(tra4);
      [l4,an4]=size(t4);
   end
      end
    end
    traa=['t1';'t2';'t3';'t4'];
    for i=1:4;
       eval(['ta=' traa(i,:) ';']);
       if (~ isempty(ta))
   plot(ta(:,7),[tra(i,:),'-'])
       end
    end
    xlabel('etapa');
    ylabel('costo');
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.1.10 cotodo.m

```
function cotodo(Par1,Par2,Par3,Par4)
% coPar(Par1,..,4) pinta Parametros juntos con ~= colores
% Permite hasta 4 matrices de Pararametros
%    function coPar(Par1,Par2,Par3,Par4)

col=['r';'g';'y';'m'];

if (nargin>=1)
   par=eval(['Par1'])
   eval(['global ' par ' ' par 'PAR']);
   argbp= par;
   argcoP=[ par 'PAR'];
   for i=2:min(nargin,4)
      par=eval(['Par',int2str(i)]);
      eval(['global ' par ' ' par 'PAR']);
      eval(par)
      argbp=[argbp ',' par];
      argcoP=[argcoP ',' par 'PAR'];
   end
   argbp
   argcoP
   eval(['musPar(' argcoP ')'])

else
   disp('ERROR.Falta matriz de parametros')
end
```

A.3.1.11 mcont.m

```
function r=contiene(trc)
%
%   function r=contiene(trtay)
%  da punto inicial ,final y costo  de las
%  trayectorias que contiene

r=[];
if (nargin>=1)
   [ft,ct]=size(trc);
   pt=4;
   while(pt<=ft)
      npt=trc(pt-3,1);

      if(ct==5)
          r=[ r
          trc(pt-2,1),trc(pt,1:2),trc(pt+npt-1,1:2),trc(pt+npt-1,5)];
       else if (ct==7)
   r=[r
     trc(pt-2,1),trc(pt,1:3),trc(pt+npt-1,1:3),trc(pt+npt-1,7)];
             end
      end
      pt=pt+npt+3;
   end
else
  echo "falta matriz trayectoria"
end
```

A.3.1.12 midsen.m

```
function Sen=midsen(tra,Mpes,Npr,Rmus,graf,Estru)
%MIDSEN mide sensibilidad de unidades dados los pesos y trajectoria
%Formato:
%function Sen=midsen(tra,Pes,Npr,Rmus,graf,Estru)
%        tra     Tryectoria de salida de la red
% Mpes    Matriz de Pesos
% Npr     Numero de puntos de prueba [100]
% Rmus    Rango de variacion del punto inicial [1e-3]
% graf    Si ~=0 representacion grafica [0]
% Estru   Estructura de la red [ [2 5 2] ]

if(nargin<2)
   error('Necesarios los 2 argumentos');
end
ranMus=1e-3;
if(nargin>=4)
   ranMus=Rmus(1);
end
npru=100;
if(nargin>=3)
   npru=Npr(1);
end
SalG=0;
if(nargin>=5)
   SalG=graf(1);
end
stru=[2 5 2];
if(nargin>=6)
```

```
   stru=Estru;
   if(min(size(stru))>1)
      error('El parametro Estru debe ser un vector');
   end
end
t=sacatn(tra);
[npes,unis]=size(Mpes);
ncap=length(stru)-1;
Sen=zeros(1,unis);
for u=1:unis
   %obtencion de pto ini y final de la unidad
   pini=t(u,1:stru(1));
   dpru=(rand(npru,stru(1))-0.5)*ranMus;
   ppru=[zeros(1,stru(1));dpru]+ones(npru+1,1)*pini;
   Ys_1=ppru';
   %obtencion de los pesos y bias de cada capa
   ptpini=1;
   for c=1:ncap
      pescap=(stru(c)+1)*stru(c+1); %pesos capa actual
      if(ptpini+pescap-1>npes)
         error('Numero de pesos no es suficiente para la estructura');
      end
      W=reshape(Mpes(ptpini:ptpini+pescap-1,u),stru(c)+1,stru(c+1))';
      B=W(:,1);
      W(:,1)=[];
      Ys=tansig(W*Ys_1,B);
      ptpini=ptpini+pescap;
      Ys_1=Ys;
   end
   if(ptpini~=(npes+1))
      error('Sobran pesos para esta estructura');
   end
   pres=(Ys*4)';
   if(SalG)
      clg;
      plot(ppru(:,1),ppru(:,2),'+',pres(:,1),pres(:,2),'o');
      axis('equal');
      hold on
      plot([ppru(1,1);pres(1,1)],[ppru(1,2);pres(1,2)],'w')
      for i=2:length(ppru)
  plot([ppru(i,1);pres(i,1)],[ppru(i,2);pres(i,2)],'r')
      end
      pause
   end
   %pasamos a calcular la sensibilidad
   dres=pres(2:npru+1,:)-ones(npru,1)*pres(1,:);

   Sens= dist(dres) ./ dist(dpru);
   Sen(u)=mean(Sens);
end
```

A.3.1.13 midsenC.m

```
function Sen=midsenC(tra,Mpes,Npr,Rmus,Estru)
%MIDSEN mide sensibilidad de las capas dados los pesos y trajectoria
%Formato:
%function Sen=midsenC(tra,Pes,Npr,Rmus,Nc1)
%        tra     Tryectoria de salida de la red
% Pes     Matriz de Pesos
% Npr     Numero de puntos de prueba [100]
% Rmus    Rango de variacion del punto inicial [1e-3]
% Estru   Estructura de la red [ [2 5 2] ]
%IMPORTANTE: No se aplica factor escala de ultima capa

if(nargin<2)
   error('Necesarios los 2 argumentos');
end
ranMus=1e-3;
if(nargin>=4)
   ranMus=Rmus(1);
end
npru=100;
if(nargin>=3)
   npru=Npr(1);
end
stru=[2 5 2];
if(nargin>=5)
   stru=Estru;
   if(min(size(stru))>1)
      error('El parametro Estru debe ser un vector');
   end
end
t=sacatn(tra);
[npes,unis]=size(Mpes);
ncap=length(stru)-1;
Sen=zeros(ncap,unis);  %1a fiala 1a capa. 2a fila 2a capa
for u=1:unis
   pini=t(u,1:stru(1));
   ptpini=1;
   for c=1:ncap
      %obtencion de los pesos y bias de cada capa
      pescap=(stru(c)+1)*stru(c+1); %pesos capa actual
      if(ptpini+pescap-1>npes)
         error('Numero de pesos no es suficiente para la estructura');
      end
      W=reshape(Mpes(ptpini:ptpini+pescap-1,u),stru(c)+1,stru(c+1))';
      B=W(:,1);
      W(:,1)=[];

      %obtencion de pto ini y final de la unidad
      dpru=(rand(npru,stru(c))-0.5)*ranMus;
      ppru=[zeros(1,stru(c));dpru]+ones(npru+1,1)*pini;
      pres=tansig(W*ppru',B)';
      dres=pres(2:npru+1,:)-ones(npru,1)*pres(1,:);
      Sens= dist(dres) ./ dist(dpru);
      Sen(c,u)=mean(Sens);

      ptpini=ptpini+pescap;
      pini=pres(1,:);
   end
   if(ptpini~=(npes+1))
      error('Sobran pesos para esta estructura');
   end
```

```
end
```

A.3.1.14 numtra.m

```
function n=numtra(tra)
% NUMTRA devuelve el numero de trayectorias en matriz empaquetada tra.
%  function n=numtra(tra)
%
if (nargin>=1)
   [nf,nc]=size(tra);
   pt=4;  n=0;
   while (pt<nf)
     n=n+1;
     np=tra(pt-3,1);
     pta=pt;
     pt=pt+np+3;
   end
else
  error(' Falta matriz de entrada')
end
```

A.3.1.15 sacan.m

```
function ts=sacan(tra,n)
% Saca la n-sima trayectoria de la matriz empaquetada tra.
%   DEJANDO las primeras lineas de parametros.
%      function ts=sacatn(tra,n)
%
if (nargin>=1)
   ts=[];
   if (nargin<2)
      n=1;
   end
   [nf,nc]=size(tra);
   pt=4;  i=0;
   while ((pt<nf) & (i~=n))
     i=i+1;
     np=tra(pt-3,1);
     pta=pt;
     pt=pt+np+3;
   end
   if (n==i & pt-4<=nf)
      ts=tra(pta-3:pt-4,:);
   end
else
  error(' Falta matriz de entrada')
end
```

A.3.1.16 sacatn.m

```
function ts=sacatn(tra,n)
% Saca la n-sima trayectoria de la matriz empaquetada tra.
%   eliminando las lineas de parametros.
%      function ts=sacatn(tra,n)
%
if (nargin>=1)
   ts=[];
   if (nargin<2)
      n=1;
   end
   [nf,nc]=size(tra);
   pt=4;  i=0;
   while ((pt<nf) & (i~=n))
     i=i+1;
     np=tra(pt-3,1);
     pta=pt;
     pt=pt+np+3;
   end
   if (n==i & pt-4<=nf)
      ts=tra(pta:pt-4,:);
   end
else
  error(' Falta matriz de entrada')
end
```

A.3.1.17 traza.m

```
function traza(mt,opc)
% Muestra trayectoria (desempaquetada) de mt
%    function traza(mt,opc)
%
if (nargin<2)
   opc='g';
end
if (nargin>=1)
  plot(mt(:,1),mt(:,2),opc)
  xlabel('x1')
  ylabel('x2')
else
   error('Falta matriz trayectoria')
end
```

A.3.1.18 trazatc.m

```
function trazatc(mt,opc)
% Muestra tiempo comando de trayectoria (desempaquetada) en  mt
%    function trazatc(mt,opc)
%
if (nargin<2)
   opc='g';
end
```

```
if (nargin>=1)
  pulsos(mt(:,4),mt(:,3),opc)
  xlabel('Tiempo')
  ylabel('Comando')
else
   error('Falta matriz trayectoria')
end
```

A.3.1.19 valsin.m

```
function [vM,vm,vMnoB,vmnoB]=valsin(Mpes,Estru)
% [vM,vm]=valsin(Mpes)  los valores singulares de las capas [max,min]
if (nargin<1)
   error('Falta matriz de pesos');
end
stru=[2 5 2]; %valor por defecto para la estructura
if(nargin>=2)
   stru=Estru;
   if(min(size(stru))>1)
      error('El parametro Estru debe ser un vector');
   end
end
[puni,Nuni]=size(Mpes);
ncap=length(stru)-1;
vM=zeros(ncap,Nuni);
vm=vM; vMnoB=vM; vmnoB=vm;
for u=1:Nuni
   ptpini=1;
   for c=1:ncap
      pescap=(stru(c)+1)*stru(c+1); %pesos capa actual
      if(ptpini+pescap-1>puni)
  error('Numero de pesos no es suficiente para la estructura');
      end
      C=reshape(Mpes(ptpini:ptpini+pescap-1,u),stru(c)+1,stru(c+1))';
      vs=svd(C);
      vM(c,u)=max(vs);
      vm(c,u)=min(vs);
      C(:,1)=[];  %Quitamos el bias
      vs=svd(C);
      vMnoB(c,u)=max(vs);
      vmnoB(c,u)=min(vs);
      ptpini=ptpini+pescap;
   end
   if(ptpini~=(puni+1))
      error('Sobran pesos para esta estructura');
   end
end
```

A.3.2 Representación Gráfica

A.3.2.1 coPar.m

```
function coPar(Par1,Par2,Par3,Par4)
% coPar(Par1,..,4) pinta Parametros juntos con ~= colores
% Permite hasta 4 matrices de Pararametros
%    function coPar(Par1,Par2,Par3,Par4)

col=['r';'g';'y';'m'];

if (nargin>=1)
   musPar(Par1,col(1));
   if (nargin>=2)
      musPar(Par2,col(2));
      if (nargin>=3)
  musPar(Par3,col(3));
  if (nargin>=4)
     musPar(Par4,col(4));
  end
      end
   end
else
  disp('ERROR.Falta matriz de parametros')
end
```

A.3.2.2 coTbp.m

```
function coTbp(T,tra1,tra2,tra3,tra4)
% pinta trayectorias juntas pero con  distintos colores o trazos
% Permite hasta 4 trayectorias
%    function coTbp(T,tra1,tra2,tra3,tra4)
%  T  -> tipo: 1-color 2-trazo
%  trai -> trayectoria con cabecera, si existen varias, la primera

t1=[]; t2=[]; t3=[]; t4=[];
l1=1;  l2=1;  l3=1;  l4=1;
MTraz=['r   ';'g   ';'c5  ';'c6  '
       '-g  ';'--g ';'-.g ';':g  '
       '-r  ';'--r ';'-.r ';':r  '
       'xr  ';'xg  ';'c5 x';'c6 x'
       'or  ';'og  ';'o c5';'o c6'];
if (nargin>=1)
   if (length(T)>1)
      error('El primer par-metro ha de ser el tipo');
   else
      ini=rem(T-1,5)*4+1;
      tra=MTraz(ini:ini+3,:);
   end
   if (nargin>=2)
      t1=sacatn(tra1);
      [l1,an1]=size(t1);
      if (nargin>=3)
  t2=sacatn(tra2);
  [l2,an1]=size(t2);
```

```
   if (nargin>=4)
      t3=sacatn(tra3);
      [l3,an3]=size(t3);
      if (nargin>=5)
         t4=sacatn(tra4);
         [l4,an4]=size(t4);
      end
   end
       end
       if (~ isempty(t1))
   maxi=max([ t1' t2' t3' t4']');
   mini=min([ t1' t2' t3' t4']');
   ml=max([l1 l2 l3 l4]);

   clg
   subplot(211);
   hold off
   axis([mini(1) maxi(1) mini(2) maxi(2)])

   traza(t1,tra(1,:))
   hold on
   if (~ isempty(t2))
      traza(t2,tra(2,:))
      if (~ isempty(t3))
         traza(t3,tra(3,:))
         if (~ isempty(t4))
            traza(t4,tra(4,:))
         end
      end
   end


   subplot(223)
   hold off
   axis([1 ml mini(3) maxi(3)])
   plot(1:l1,t1(:,3),tra(1,:))
   hold on
   if (~ isempty(t2))
      plot(1:l2,t2(:,3),tra(2,:))
      if (~ isempty(t3))
         plot(1:l3,t3(:,3),tra(3,:))
         if (~ isempty(t4))
            plot(1:l4,t4(:,3),tra(4,:))
         end
      end
   end
   xlabel('Etapa');
   ylabel('Comando');

   subplot(224)
   hold off
   axis([1 ml mini(4) maxi(4)])
   plot(1:l1,t1(:,4),tra(1,:))
   hold on
   if (~ isempty(t2))
      plot(1:l2,t2(:,4),tra(2,:))
      if (~ isempty(t3))
         plot(1:l3,t3(:,4),tra(3,:))
         if (~ isempty(t4))
            plot(1:l4,t4(:,4),tra(4,:))
         end
      end
   end
   xlabel('Etapa');
   ylabel('Tiempo');

   hold off
   axis;
       else
   disp('No hay trayectorias')
      end
    else
      disp('ERROR.Falta matriz de trayectorias')
    end
else
   error('Faltan los parametros de entrada');
end
```

A.3.2.3 cobp.m

```
function cobp(tra1,tra2,tra3,tra4)
% pinta trayectorias juntas
% Permite hasta 4 trayectorias
%    function cobp(tra1,tra2,tra3,tra4)
%  trai -> trayectoria con cabecera, si existen varias, la primera

tra=['r';'g';'y';'m'];

t1=[]; t2=[]; t3=[]; t4=[];
l1=1;  l2=1;  l3=1;  l4=1;
if (nargin>=1)
   t1=sacatn(tra1);
   [l1,an1]=size(t1);
   if (nargin>=2)
      t2=sacatn(tra2);
      [l2,an1]=size(t2);
      if (nargin>=3)
   t3=sacatn(tra3);
   [l3,an3]=size(t3);
   if (nargin>=4)
      t4=sacatn(tra4);
      [l4,an4]=size(t4);
   end
      end
   end
   if (~ isempty(t1))
      maxi=max([ t1' t2' t3' t4']');
      mini=min([ t1' t2' t3' t4']');
      ml=max([l1 l2 l3 l4]);

      clg
      subplot(211);
      hold off
```

```
        axis([mini(1) maxi(1) mini(2) maxi(2)])

        traza(t1,tra(1,:))
        hold on
        if (~ isempty(t2))
    traza(t2,tra(2,:))
    if (~ isempty(t3))
       traza(t3,tra(3,:))
       if (~ isempty(t4))
          traza(t4,tra(4,:))
       end
    end
        end


        subplot(223)
        hold off
        axis([1 ml mini(3) maxi(3)])
        plot(1:l1,t1(:,3),tra(1,:))
        hold on
        if (~ isempty(t2))
    plot(1:l2,t2(:,3),tra(2,:))
    if (~ isempty(t3))
       plot(1:l3,t3(:,3),tra(3,:))
       if (~ isempty(t4))
          plot(1:l4,t4(:,3),tra(4,:))
       end
    end
        end
        xlabel('Etapa');
        ylabel('Comando');

        subplot(224)
        hold off
        axis([1 ml mini(4) maxi(4)])
        plot(1:l1,t1(:,4),tra(1,:))
        hold on
        if (~ isempty(t2))
    plot(1:l2,t2(:,4),tra(2,:))
    if (~ isempty(t3))
       plot(1:l3,t3(:,4),tra(3,:))
       if (~ isempty(t4))
          plot(1:l4,t4(:,4),tra(4,:))
       end
    end
        end
        xlabel('Etapa');
        ylabel('Tiempo');

        hold off
        axis;
     else
        disp('No hay trayectorias')
     end
else
   disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.4 cobpj.m

```
function comij(trac,ind)
%function comij(trac,ind) Muestra juntas trayectorias en trac
%  Si se indica ind solo aquellas cuyo numero aparece en ind


if (nargin>=1)
    clg
    subplot(211);
    hold on;
    xlabel('X1');
    ylabel('X2');
    subplot(223);
    hold on;
    xlabel('etapa');
    ylabel('Comando');
    subplot(224);
    hold on;
    xlabel('etapa');
    ylabel('tiempos');
    if(nargin<2)
       ind=1:numtra(trac);
    end
    for i=1:length(ind(:))
       ta=sacatn(trac,i);
       if(~isempty(ta))
   color=['c',int2str(i+1)];
   subplot(211);
   plot(ta(:,1),ta(:,2),[color,'-'])
   subplot(223);
   plot(ta(:,3),[color,'-'])
   subplot(224);
   plot(ta(:,4),color)
       end
    end
else
   disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.5 comi.m

```
function comi(tra1,tra2,tra3,tra4,tra5)
%comi([numtra,]tra1,tra2,tra3,tra4) pinta juntas las trayectorias
% pinta trayectorias juntas
% Permite hasta 4 trayectorias
%  trai -> trayectoria con cabecera
%  numtra-> Permite elegir tra. de las empaquetadas

tra=['r';'g';'y';'m'];

t1=[]; t2=[]; t3=[]; t4=[];
```

```
if (nargin>=1)
   if(length(tra1)==1)
      traa=2:nargin;
      numtr=tra1;
   else
      traa=1:nargin;
      numtr=1;
   end
   clg
   subplot(221);
   hold on;
   xlabel('etapa');
   ylabel('Niveles');
   subplot(222);
   hold on;
   xlabel('etapa');
   ylabel('Inc Temp');
   subplot(223);
   hold on;
   xlabel('etapa');
   ylabel('flujos');
   subplot(224);
   hold on;
   xlabel('etapa');
   ylabel('tiempos');
   i=0;
   for j=traa;
      i=i+1;
      comando=['ta=tra' int2str(j) ';'];
      eval(comando);
      ta=sacatn(ta,numtr);
      if (~ isempty(ta))
  subplot(221);
  plot(ta(:,1),[tra(i,:),'-'])
  plot(ta(:,2),[tra(i,:),'--'])
  subplot(222);
  plot(ta(:,3),tra(i,:))
  subplot(223);
  plot(ta(:,4),[tra(i,:),'-'])
  plot(ta(:,5),[tra(i,:),'--'])
  subplot(224);
  plot(ta(:,6),tra(i,:))
      end
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.6 comij.m

```
function comij(trac,ind)
%function comij(trac,ind) Muestra juntas trayectorias en trac
%  Si se indica ind solo aquellas cuyo numero aparece en ind


if (nargin>=1)
   clg
   subplot(221);
   hold on;
   xlabel('etapa');
   ylabel('Niveles');
   subplot(222);
   hold on;
   xlabel('etapa');
   ylabel('Inc Temp');
   subplot(223);
   hold on;
   xlabel('etapa');
   ylabel('flujos');
   subplot(224);
   hold on;
   xlabel('etapa');
   ylabel('tiempos');
   if(nargin<2)
      ind=1:numtra(trac);
   end
   for i=1:length(ind(:))
      ta=sacatn(trac,ind(i));
      if(~isempty(ta))
  color=['c',int2str(i+1)];
  subplot(221);
  plot(ta(:,2),[color,'--'])
  plot(ta(:,1),[color,'-'])
  subplot(222);
  plot(ta(:,3),color)
  subplot(223);
  plot(ta(:,4),[color,'-'])
  plot(ta(:,5),[color,'--'])
  subplot(224);
  plot(ta(:,6),color)
      end
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.7 comip.m

```
function comip(tra1,tra2,tra3,tra4,tra5)
%comi([numtra,]tra1,tra2,tra3,tra4) pinta juntas las trayectorias
% pinta trayectorias juntas
% Permite hasta 4 trayectorias
%  trai -> trayectoria con cabecera
%  numtra-> Permite elegir tra. de las empaquetadas

tra=['r';'g';'y';'m'];

t1=[]; t2=[]; t3=[]; t4=[];
if (nargin>=1)
   if(size(tra1,1)==1)
```

```
      traa=2:nargin;
      traele=tra1;
   else
      traa=1:nargin;
      traele=1:numtra(tra1);
   end
   for numtr=traele
      disp('*** Calculando ***');
      clg
      subplot(221);
      hold on;
      xlabel('etapa');
      ylabel('Niveles');
      subplot(222);
      hold on;
      xlabel('etapa');
      ylabel('Inc Temp');
      subplot(223);
      hold on;
      xlabel('etapa');
      ylabel('flujos');
      subplot(224);
      hold on;
      xlabel('etapa');
      ylabel('tiempos');
      i=0;
      for j=traa;
   i=i+1;
   comando=['ta=tra' int2str(j) ';'];
   eval(comando);
   ta=sacatn(ta,numtr);
   if (~ isempty(ta))
      subplot(221);
      plot(ta(:,1),[tra(i,:),'-'])
      plot(ta(:,2),[tra(i,:),'--'])
      subplot(222);
      plot(ta(:,3),tra(i,:))
      subplot(223);
      plot(ta(:,4),[tra(i,:),'-'])
      plot(ta(:,5),[tra(i,:),'--'])
      subplot(224);
      plot(ta(:,6),tra(i,:))
      disp([ta(1,1:3) ta(size(ta,1),1:3) ta(size(ta,1),size(ta,2))]);
   end
      end
      drawnow;
      disp('Pulsa para seguir');
      pause;
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.8 como.m

```
function como(tra1,tra2,tra3,tra4,tra5)
%como([numtra,]tra1,tra2,tra3,tra4) pinta juntas las trayectorias
% pinta trayectorias juntas del sistema movil Hilare
% Permite hasta 4 trayectorias
%  trai -> trayectoria con cabecera
%  numtra-> Permite elegir tra. de las empaquetadas

tra=['r';'g';'y';'m'];

t1=[]; t2=[]; t3=[]; t4=[];
if (nargin>=1)
   if(length(tra1)==1)
      traa=2:nargin;
      numtr=tra1;
   else
      traa=1:nargin;
      numtr=1;
   end
   clg
   subplot(221);
   hold on;
   xlabel('X1');
   ylabel('X2');
   subplot(222);
   hold on;
   xlabel('etapa');
   ylabel('Grados');
   subplot(223);
   hold on;
   xlabel('etapa');
   ylabel('Comandos');
   subplot(224);
   hold on;
   xlabel('etapa');
   ylabel('tiempos');
   i=0;
   for j=traa;
      i=i+1;
      comando=['ta=tra' int2str(j) ';'];
      eval(comando);
      ta=sacatn(ta,numtr);
      if (~ isempty(ta))
   subplot(221);
   plot(ta(:,1),ta(:,2),[tra(i,:),'-'])
   subplot(222);
   plot(ta(:,3)*180/pi,tra(i,:))
   subplot(223);
   plot(ta(:,4),[tra(i,:),'-'])
   plot(ta(:,5),[tra(i,:),'--'])
   subplot(224);
   plot(ta(:,6),tra(i,:))
      end
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.9 comoj.m

```
function com0j(trac,ind)
%comoj(trac,ind) Muestra juntas trayectorias en trac (sist hilare)
%  Si se indica ind solo aquellas cuyo numero aparece en ind


if (nargin>=1)
   clg
   subplot(221);
   hold on;
   xlabel('X1');
   ylabel('X2');
   subplot(222);
   hold on;
   xlabel('etapa');
   ylabel('Grados');
   subplot(223);
   hold on;
   xlabel('etapa');
   ylabel('Comandos');
   subplot(224);
   hold on;
   xlabel('etapa');
   ylabel('tiempos');
   if(nargin<2)
      ind=1:numtra(trac);
   end
   for i=1:length(ind(:))
      ta=sacatn(trac,ind(i));
      if(~isempty(ta))
  color=['c',int2str(i+1)];
  subplot(221);
  plot(ta(:,1),ta(:,2),[color,'-'])
  subplot(222);
  plot(ta(:,3)*180/pi,color)
  subplot(223);
  plot(ta(:,4),[color,'-'])
  plot(ta(:,5),[color,'--'])
  subplot(224);
  plot(ta(:,6),color)
     end
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.10 comojp.m

```
function com0j(trac,ind)
%comoj(trac,ind) Muestra juntas trayectorias en trac (sist hilare)
%  Si se indica ind solo aquellas cuyo numero aparece en ind


if (nargin>=1)
   clg
   subplot(221);
   hold on;
   xlabel('X1');
   ylabel('X2');
   subplot(222);
   hold on;
   xlabel('etapa');
   ylabel('Grados');
   subplot(223);
   hold on;
   xlabel('etapa');
   ylabel('Comandos');
   subplot(224);
   hold on;
   xlabel('etapa');
   ylabel('tiempos');
   if(nargin<2)
      ind=1:numtra(trac);
   end
   for i=1:length(ind(:))
      ta=sacatn(trac,ind(i));
      if(~isempty(ta))
  color=['c',int2str(i+1)];
  subplot(221);
  plot(ta(:,1),ta(:,2),[color,'-'])
  subplot(222);
  plot(ta(:,3)*180/pi,color)
  subplot(223);
  bar(ta(:,4),[color,'-'])
  bar(ta(:,5),[color,'--'])
  subplot(224);
  bar(ta(:,6),color)
     end
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.11 comp.m

```
function comp(tra1,tra2,tra3,tra4)
% pinta trayectorias juntas pero con  distintos trazos
% Permite hasta 4 trayectorias
%    function comp(tra1,tra2,tra3,tra4)
%  trai -> trayectoria con cabecera, si existen varias, la primera

t1=[]; t2=[]; t3=[]; t4=[];
if (nargin>=1)
   t1=sacatn(tra1);
   maxt=sum(t1(:,4));
   if (nargin>=2)
      t2=sacatn(tra2);
```

```
      maxt=max([maxt sum(t2(:,4))]);
      if (nargin>=3)
         t3=sacatn(tra3);
         maxt=max([maxt sum(t3(:,4))]);
         if (nargin>=4)
            t4=sacatn(tra4);
            maxt=max([maxt sum(t4(:,4))]);
         end
      end
   end
   if (~ isempty(t1))
      maxi=max([ t1' t2' t3' t4']');
      mini=min([ t1' t2' t3' t4']');

      clg
      subplot(211);
      hold off
      axis([mini(1) maxi(1) mini(2) maxi(2)])

      traza(t1,'r')
      hold on
      if (~ isempty(t2))
         traza(t2,'g')
         if (~ isempty(t3))
            traza(t3,'y')
            if (~ isempty(t4))
               traza(t4,'m')
            end
         end
      end

      subplot(212)
      hold off
      axis([0 maxt mini(3) maxi(3)])
      trazatc(t1,'r')
      hold on
      if (~ isempty(t2))
         trazatc(t2,'g')
         if (~ isempty(t3))
            trazatc(t3,'y')
            if (~ isempty(t4))
               trazatc(t4,'m')
            end
         end
      end

      hold off
      axis;
   else
      disp('No hay trayectorias')
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.12 compj.m

```
function compj(trac,ind)
%function comij(trac,ind) Muestra juntas trayectorias en trac
%  Si se indica ind solo aquellas cuyo numero aparece en ind


if (nargin>=1)
   clg
   subplot(211);
   hold on;
   xlabel('X1');
   ylabel('X2');
   subplot(212);
   hold on;
   xlabel('tiempo');
   ylabel('comando');
   if(nargin<2)
      ind=1:numtra(trac);
   end
   for i=1:length(ind(:))
      ta=sacatn(trac,i);
      if(~isempty(ta))
  color=['c',int2str(i+1)];
  subplot(211);
  plot(ta(:,1),ta(:,2),[color,'-'])
  subplot(212);
  pulsos(ta(:,4),ta(:,3),[color,'-'])
      end
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.13 comti.m

```
function comti(tra1,tra2,tra3,tra4)
% pinta trayectorias juntas frente al tiempo NO TIEMPOS NEGATIVOS
% Permite hasta 4 trayectorias
%    function cobp(tra1,tra2,tra3,tra4)
%  trai -> trayectoria con cabecera, si existen varias, la primera

tra=['r';'g';'y';'m'];

t1=[]; t2=[]; t3=[]; t4=[];
l1=1;  l2=1;  l3=1;  l4=1;
if (nargin>=1)
   t1=sacatn(tra1);
   [l1,an1]=size(t1);
   if (nargin>=2)
      t2=sacatn(tra2);
      [l2,an1]=size(t2);
      if (nargin>=3)
  t3=sacatn(tra3);
  [l3,an3]=size(t3);
```

```
   if (nargin>=4)
      t4=sacatn(tra4);
      [l4,an4]=size(t4);
   end
       end
    end
    clg
    subplot(221);
    hold on;
    xlabel('tiempo');
    ylabel('Niveles');
    subplot(222);
    hold on;
    xlabel('tiempo');
    ylabel('Inc Temp');
    subplot(223);
    hold on;
    xlabel('tiempo');
    ylabel('flujos');
    subplot(224);
    hold on;
    xlabel('tiempo');
    ylabel('Calor');
    traa=['t1';'t2';'t3';'t4'];
    for i=1:4;
       eval(['ta=' traa(i,:) ';']);
       if (~ isempty(ta))
   nfil=size(ta,1);
   time=cumsum([0; ta(1:nfil-1,6)]);
   subplot(221);
   plot(time,ta(:,1),[tra(i,:),'-'])
   plot(time,ta(:,2),[tra(i,:),'--'])
   subplot(222);
   plot(time,ta(:,3),tra(i,:))
   subplot(223);
   pulsos(ta(:,6),ta(:,4),[tra(i,:),'-'])
%  plot(time(1:nfil-1),ta(1:nfil-1,4),[tra(i,:),'-'])
%  plot(time,ta(:,5),[tra(i,:),'--'])
   subplot(224);
   pulsos(ta(:,6),ta(:,5),[tra(i,:),'--'])
%  plot(time(1:nfil-1),ta(1:nfil-1,5),[tra(i,:),'--'])
%  plot(time,ta(:,6),tra(i,:))
       end
    end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.14 muesj.m

```
function muesj(tra1)
% pinta juntas  las 4 primeras trayectorias de la matriz
%    function muesj(tra1)
%  trai -> trayectoria con cabecera

tra=['r';'g';'y';'m'];

t1=[]; t2=[]; t3=[]; t4=[];
l1=1;  l2=1;  l3=1;  l4=1;
if (nargin>=1)
   t1=sacatn(tra1,1);
   [l1,an1]=size(t1);
   t2=sacatn(tra1,2);
   [l2,an2]=size(t2);
   t3=sacatn(tra1,3);
   [l3,an3]=size(t3);
   t4=sacatn(tra1,4);
   [l4,an4]=size(t4);
   if (~ isempty(t1))
      maxi=max([ t1' t2' t3' t4']');
      mini=min([ t1' t2' t3' t4']');
      ml=max([l1 l2 l3 l4]);

      clg
      subplot(211);
      hold off
      axis([mini(1) maxi(1) mini(2) maxi(2)])

      traza(t1,tra(1,:))
      hold on
      if (~ isempty(t2))
   traza(t2,tra(2,:))
   if (~ isempty(t3))
      traza(t3,tra(3,:))
      if (~ isempty(t4))
         traza(t4,tra(4,:))
      end
   end
       end


       subplot(223)
       hold off
       axis([1 ml mini(3) maxi(3)])
       plot(1:l1,t1(:,3),tra(1,:))
       hold on
       if (~ isempty(t2))
   plot(1:l2,t2(:,3),tra(2,:))
   if (~ isempty(t3))
      plot(1:l3,t3(:,3),tra(3,:))
      if (~ isempty(t4))
         plot(1:l4,t4(:,3),tra(4,:))
      end
   end
       end
       xlabel('Etapa');
       ylabel('Comando');

       subplot(224)
       hold off
       axis([1 ml mini(4) maxi(4)])
       plot(1:l1,t1(:,4),tra(1,:))
       hold on
```

```
      if (~ isempty(t2))
   plot(1:l2,t2(:,4),tra(2,:))
   if (~ isempty(t3))
      plot(1:l3,t3(:,4),tra(3,:))
      if (~ isempty(t4))
         plot(1:l4,t4(:,4),tra(4,:))
      end
   end
      end
      xlabel('Etapa');
      ylabel('Tiempo');

      hold off
      axis;
   else
      disp('No hay trayectorias')
   end
else
  disp('ERROR.Falta matriz de trayectorias')
end
```

A.3.2.15 musPar.m

```
function musPar(MPar,co)
%musPar(MPar)  muestra ValSin y Sensib. en 2 garficas separadas

if(nargin<1)
   error('Falta matriz de parametros')
end
if(size(MPar,1)~=11)
   error('Solo preparado para redes de 2 capas => MPar 11 filas');
end
c='r';
if(nargin>=2)
   c=co;
end
subplot(211); hold on;
xlabel('unidad'); ylabel('Valor sigunlar');
l=1:size(MPar,2);   %columnas de MPar
plot(l,MPar(1,:)',c,l,MPar(2,:)',[c,'--'],  ...
   l,MPar(3,:)',[c,'-.'],l,MPar(4,:)',[c,':']);
plot(l,MPar(5,:)',['x',c],l,MPar(6,:)',[c,'x'],  ...
   l,MPar(7,:)',[c,'x'],l,MPar(8,:)',[c,'x']);
plot(l,MPar(5,:)',c,l,MPar(6,:)',[c,'--'],  ...
   l,MPar(7,:)',[c,'-.'],l,MPar(8,:)',[c,':']);
%plot(l,MPar(1,:).*MPar(2,:),[c,'*'])

title('_ Max c1; -- Max c2; -. Min c1; : Min c2. x idem sin Bias');

subplot(212); hold on;
xlabel('unidad'); ylabel('Sensibilidad');
plot(l,MPar(9,:)',[c,'--'])      %Capa1
plot(l,MPar(10,:)/4,[c,':'])     %Capa 2
plot(l,MPar(11,:)/4,[c])         %Completa
%plot(l,MPar(9,:).*MPar(10,:)/4,[c,'*'])
%plot(l,MPar(1,:)/2,[c,'+']);
%plot(l,MPar(5,:)/2,[c,'+']);
%plot(l,MPar(2,:)/10.6,[c,'o']);
%plot(l,MPar(6,:)/10.6,[c,'o']);

title('_ Completa; -- Capa 1; : Capa 2');
hold on;
```

A.3.2.16 musparco.m

```
function musPar(MPar,co)
%musPar(MPar)  muestra ValSin y Sensib. en 2 garficas separadas

if(nargin<1)
   error('Falta matriz de parametros')
end
if(size(MPar,1)~=11)
   error('Solo preparado para redes de 2 capas => MPar 11 filas');
end
c='r';
if(nargin>=2)
   c=co;
end
subplot(211); hold on;
xlabel('unidad'); ylabel('Valor sigunlar');
l=1:size(MPar,2);   %columnas de MPar
plot(l,MPar(1,:)',c,l,MPar(2,:)',[c,'--'],  ...
   l,MPar(3,:)',[c,'-.'],l,MPar(4,:)',[c,':']);
plot(l,MPar(5,:)',['o',c],l,MPar(6,:)',[c,'+'],  ...
   l,MPar(7,:)',[c,'x'],l,MPar(8,:)',[c,'*']);
%plot(l,MPar(1,:).*MPar(2,:),[c,'*'])

subplot(212); hold on;
xlabel('unidad'); ylabel('Sensibilidad');
plot(l,MPar(9,:),c)
plot(l,MPar(10,:)/4,[c,'--'])
plot(l,MPar(11,:)/4,[c,':'])
plot(l,MPar(9,:).*MPar(10,:)/4,[c,'*'])
plot(l,MPar(1,:)/2,[c,'+']);
plot(l,MPar(5,:)/2,[c,'+']);
plot(l,MPar(2,:)/10.6,[c,'o']);
plot(l,MPar(6,:)/10.6,[c,'o']);
hold on;
```

A.3.2.17 pintor.m

```
function pintor(te,nt)
%pintor(tar,nt)  Pinta trayectoria con direcciones y sentidos de avance

if nargin>=1
   if nargin<2
      nt=1;
```

```
   end
   tra=sacatn(te,nt(1));
   clg
   subplot(211);
   plot(tra(:,1),tra(:,2),'r')
   hold on;
   xlabel('X1'); ylabel('X2');
   title('Trayectoria con vector orientacion');
   plot(tra(:,1),tra(:,2),'xr')
   ejes=axis;
   minl=min([ejes(2)-ejes(1);ejes(4)-ejes(3)]);
   lonv=minl/10;
   for p=tra(:,1:3)'
      pd=[p(1)+lonv*cos(p(3)) p(2)+lonv*sin(p(3))];
      plot([p(1) pd(1)],[p(2) pd(2)],'g')
   end
   subplot(223);
   bar(tra(1:size(tra,1)-1,4));
   hold on;
   xlabel('Etapas');
   ylabel('Avance');
   subplot(224);
   bar(tra(1:size(tra,1)-1,6));
   hold on;
   xlabel('Etapas');
   ylabel('Tiempos');
else
   error('Falta matriz de trayectorias empaquetadas');
end
drawnow
```

A.3.2.18 pulsos.m

```
function pulsos(m1,m2,op)
% pinta m2 frente a m1 como pulsos
%    function pulsos(m1,m2,opcionColor)
if (nargin >= 2)
   l1=size(m1); l2=size(m2);
   if (nargin>=3)
      color=op;
   else
      color='g';
   end
   if (l1==l2)
      ti=zeros(l1,1);
      ti=cumsum(m1);
      ti2=zeros(l1*2,1);
      ti2(2:2:l1*2)=ti;  ti2(3:2:l1*2-1)=ti(1:l1-1);
      mt2=zeros(l1*2,1);
      mt2(1:2:l1*2-1)=m2;  mt2(2:2:l1*2)=m2;
      plot(ti2,mt2,color)
   else
      error(' Matrices han de ser de igual logitud')
   end
else
   error(' Faltan parametros de entrada')
end
```

A.3.3 Otros

A.3.3.1 dist.m

```
function dse=dist(M)
%DIST(M) calcula la norma euclidea de los vectores fila de M


[nf,nc]=size(M);
dse=[];
for v=M'   %se iran tomando las filas de M
   dse=[dse;norm(v)];
end
```

A.3.3.2 entre1BP.m

```
function [p]=entre1BP(tt,nneu,nentre)
%entre1BP()

if(nargin<1)
   error('Falta trayectoria de entrada');
end
if(nargin<2)
   nneu=5;
end
if(nargin<3)
   nentre=300;
end

[np,nve]=size(tt);
for i=1:np-1
%    pause;
    P=tt(i,:)';
    R=nve;
    Q=1;
    S1=nve;
    T=(tt(i+1,:)/4)';

    %inicializamos los pesos
    [W1,B1]=nwtan(S1,R);

    TP=[nentre nentre 0.00001 0.001 1.05 0.7 0.95 1.04];
    [W1,B1]=trainbpx(W1,B1,'tansig',P,T,TP);

    t1=[B1,W1]';
```

```
    p(:,i)=t1(:);
end
```

A.3.3.3 entre2BP.m

```
function [p]=entre2BP(tt,nneu,nentre,ntr,ransal,funsal)
%Entrena serie de redes para que sigan los puntos de la trayectoria
% introducida [p]=entre2BP(tt,nneu,nentre,ntr,ransal)
%  Entrada:  tt   puntos trayectoria(length(tt)-1= no. de redes)
%      nneu   no. de neuronas de capa oculta (por defcto 5)
%      nentre no. maximo de entrenamientos (por defecto 500)
%      ntr    intervalo presentacion (<0 conpausa)(deff =nentre)
%             ransal vector con rangos de cada salida (por defecto 4)
%      funsa  funciones de salida para la ultima capa

if(nargin<1)
   error('Falta trayectoria de entrada');
end
if(nargin<2)
   nneu=5;
end
if(nargin<3)
   nentre=500;
end
if(nargin<4)
   ntr=nentre;
end
if(nargin<6)
   funsal='tansig';
end

[np,nve]=size(tt);
for i=1:np-1
    if nentre<0
       disp('pulsa para seguir');
       pause;
       clc
       tt(i:i+1,:)
    end
    P=tt(i,:)';
    R=nve;
    Q=1;
    S1=nneu;
    S2=nve;
    if(nargin<5)
 T=(tt(i+1,:)/4)';
    else
 T=tt(i+1,:)'./ransal(:);
    end


    %inicializamos los pesos
    [W1,B1]=nwtan(S1,R);
    [W2,B2]=rands(S2,S1);
    W2=W2*0.5;
    W1=W1*0.5;

    if (nargin>=3)
    TP=[ntr abs(nentre) 1e-15];
    [W1,B1,W2,B2]=trainbpx(W1,B1,'tansig',W2,B2,funsal,P,T,TP);
    else
    [W1,B1,W2,B2]=trainbpx(W1,B1,'tansig',W2,B2,funsal,P,T);
    end

    t1=[B1,W1]';
    t2=[B2,W2]';

    p(:,i)=[t1(:);t2(:)];
end
```

A.3.3.4 entretra.m

```
function [p]=entretra(tt,nneu,nentre,ntr,ransal,funsal)
%Entrena serie de redes para que sigan los puntos de la trayectoria
% introducida [p]=entre2BP(tt,nneu,nentre,ntr,ransal,funsal)
%  Entrada:  tt    trayectorias empaquetadas (cortada al ancho de VE)
%      nneu   no. de neuronas de capa oculta (por defcto 5)
%      nentre no. maximo de entrenamientos (por defecto 500)
%      ntr    intervalo presentacion (<0 conpausa)(deff =nentre)
%             ransal vector con rangos de cada salida (por defecto 4)
%      funsa  funciones de salida para la ultima capa

if(nargin<1)
   error('Falta trayectoria de entrada');
end
if(nargin<2)
   nneu=5;
end
if(nargin<3)
   nentre=500;
end
if(nargin<4)
   ntr=nentre;
end
if(nargin<6)
   funsal='tansig';
end

%Al tratarse de varias trayectorias hay que hacer entrenamiento
% de grupo.
[np,nve]=size(sacatn(tt));
nt=numtra(tt);
%separamos trayectorias y metemos en matrices separadas
for t=1:nt
   ta=sacatn(tt,t);
   eval(['tr' int2str(t) '=ta;']);
end
%definimos los tamanios de T y P
T=zeros(nve,nt);  P=T;
```

```
for i=1:np-1
    if nentre<0
       disp('pulsa para seguir');
       pause;
       clc
       tt(i:i+1,:)
    end
% P ahora de varias comumnas    P=tt(i,:)';
   for t=1:nt
      eval(['ta=tr' int2str(t) ';']);
      P(:,t)=ta(i,:)';
      T(:,t)=ta(i+1,:)';
   end
    R=nve;
    Q=1;
    S1=nneu;
    S2=nve;
    if(nargin<5)
 T=T./4;
    else
 T=T./(ransal(:)*ones(1,nt));
    end

    %inicializamos los pesos
    [W1,B1]=nwtan(S1,R);
    [W2,B2]=rands(S2,S1);
    W2=W2*0.5;
    W1=W1*0.5;

    if (nargin>=3)
    TP=[ntr abs(nentre) 1e-10];
    [W1,B1,W2,B2]=trainbpx(W1,B1,'tansig',W2,B2,funsal,P,T,TP);
    else
    [W1,B1,W2,B2]=trainbpx(W1,B1,'tansig',W2,B2,funsal,P,T);
    end

    t1=[B1,W1]';
    t2=[B2,W2]';

    p(:,i)=[t1(:);t2(:)];
end
```

Referencias

[Acos 91] L. Acosta: "Concepción y Desarrollo de Métodos, Basados en Lógica Heurística, para la Reducción Espacial/Temporal de la Programación Dinámica en Procesos de Control Óptimo Continuos/Discretos Deterministas y Estocásticos". Tesis Doctoral. Dep. de Física Fundamental y Experimental. Univ. de La laguna. Marzo de 1991.

[Acos 94] L. Acosta, A. Hamilton, L. Moreno, J.L. Sánchez, J.D. Piñeiro, J.A. Méndez: "Two Approaches to Nonlinear Systems Optimal Control by Using Neural Networks". Proc. of IEEE International Conference on Neural Networks. WCCI'94. Orlando. June-July 1994.

[Alec 90] "Control de Procesos Industriales CPI-100". ALECOP. 1990.

[Åstr 84] K.J. Åström, B. Wittenmark: "Computer Controlled Systems: Theory and Design". Prentice-Hall. 1984.

[Atha 66] M. Athans, P.L. Falb: "Optimal Control: An Introduction to the Theory and Its Applications". McGraw-Hill. 1966

[Athe 82] D.P. Atherton: "Nonlinear Control Engineering". Van Nostrand Reinhold. 1982.

[Bank 88] S.P. Banks: "Mathematical Theories on Nonlinear Systems". Prentice-Hall. 1988.

[Barn 85] S. Barnett, R.G. Cameron: "Introduction to Mathematical Control Theory. Second Edition". Caledon Press. 1985.

[Barn 92] E. Barnard: "Optimization for Training Neural Nets". IEEE Transactions on Neural Networks, vol. 3, no. 2. March, 1992.

[Bell 62] R.E. Bellman, S.E. Dreyfus: "Applied Dynamic Programming". Princeton University Press. 1962.

[Bert 87] D.P. Bertsekas: "Dynamic Programming: Deterministic and Stochastic Models". Prentice-Hall. 1987.

[Bill 84] S.A. Billings, J.O. Cray, D.H. Owens (Editors): "Nonlinear Systems Design". Peter Peregrinus. 1984.

[Cama 95] E.F. Camacho, C. Bordóns: "Model Predictive Control in the Process Industry". Springer-Verlag. 1995.

[Cich 93] A. Cichocki, R. Unbehauen: "Neural Networks for Optimization and Signal Processing". John Wiley & Sons. 1993.

[Clar 87] D.W. Clarke, C. Mohtadi, P.S. Tuffs: "Generalized Predictive Control. Part I & II". Automatica, vol. 23, no. 2, pp. 137-160. 1987

[Clar 89] D.W. Clarke, C. Mohtadi: "Properties of Generalized Predictive Control. Part I & II". Automatica, vol. 25, no. 6, pp. 859-875. 1989

[Cook 86] P.A. Cook: "Nonlinear Dynamical Systems". Prentice-Hall. 1986.

[Cowa 93] J.D. Cowan, D.H. Sharp: "Redes Neuronales e Inteligencia Artificial". en el libro "Nuevo debate sobre la inteligencia artificial. Sistemas simbólicos y Redes Neuronales", pp. 103-144. Editorial Gedisa. 1993.

[Demi 92] H. Demircioglu, P.J. Gawthrop: "Multivariable Continuous-time Generalized Predictive Control (MCGPC)". Automatica, vol. 28, no. 4, pp. 697-713. 1992.

[Demi 93] H. Demircioglu, D.W. Clarke: "Generalized Predictive Control With End-Point State Weighting". IEE proceedings-D, vol. 140, no. 4, pp. 275-282. July 1993.

[Demu 93] H. Demuth, M. Beale: "Neural Network Toolbox for Use with MATLAB". The MathWorks. 1993.

[Flet 87] R. Fletcher: "Practical Methods of Optimization (Second Edition)". John Wiley & Sons. 1987.

[Fran 90] G.F. Franklin, J.D. Powell, M.L. Workman: "Digital Control of Dynamic Systems. Second Edition". Addison-Wesley. 1990.

[Free 93] J.A. Freeman, D.M. Skapura: "Redes Neuronales. Algoritmos, Aplicaciones y Técnicas de Programación". Addison-Wesley Iberoamericana. 1993.

[Gibs 63] J.E. Gibson: "Nonlinear Automatic Control". McGraw-Hill. 1963

[Grac 92] A. Grace, A.J. Laub, J.N. Little, C.M. Thompson: "Control Systems Toolbox for Use with MATLAB". MathWorks. 1992

[Greg 95] R.L. Gregory (Editor): "Diccionario Oxford de la Mente". Alianza Editorial. 1995.

[Hami 92] A. Hamilton: "Políticas Óptimas para Sistemas No Lineales Mediante Programación Dinámica y Redes Neuronales." Memoria de Licenciatura. Universidad de La Laguna. 1992.

[Hami 95] A. Hamilton, L. Acosta, N. Marichal, J.A. Méndez, L. Moreno: "Problemas de Control Óptimo: una Aproximación Mediante Redes Neuronales". I Jornadas de Informática, pp. 235-244, Puerto de la Cruz. Julio, 1995.

[Hern 92] J.C. Hernández, J. Riera: "Modelado, Simulación y Control de un Proceso Modular de Tanques Interconectados". Proyecto fin de Diplomatura. Centro Superior de Informática. Universidad de La Laguna. Octubre 1992

[Hewl 92] "HP-UX Reference. Release 9.0, HP 9000. Sections 1, 2, 3, 4 and 1M". Hewlett-Packard. 1992.

[Hopf 82] J.J. Hopfield: "Neural Networks and Physical Systems with Emergent Collective Computational Abilities". Proc. of the National Academy of Sciences, USA, pp. 2554-2558. 1982.

[Hopf 84] J.J. Hopfield: "Neurons with Graded Response Have Collective Computational Properties Like Those of Two-States Neurons". Proc. Nat'l. Acad. Sci. USA, pp. 3088-3092. 1984

[Hunt 91] K.J. Hunt, D. Sbarbaro: "Neural Networks for Nonlinear Model Control". IEE Proceedings-D, vol. 138, no. 5. September 1991.

[Hush 93] D.R. Hush, B.G. Horne: "Progress in Supervised Neural Networks. What's New Since Lippmann?". IEEE Signal Processing Magazine, pp. 8-29. January, 1993.

[Iser 89] R. Isermann: "Digital Control Systems. Volume 1: Fundamental, Deterministic Control. Second, Revised Edition". Springer-Verlag. 1989.

[Jacq 81] R.G. Jacquot: "Modern Digital Control Systems". Marcel Dekker. 1981.

[Kail 80] T. Kailath: "Linear Systems". Prentice-Hall. 1980.

[Kern 88] B.W. Kernighan, D.M. Ritchie: "The C Programming Language. Second Edition". Prentice-Hall. 1988.

[Kouv 92] B. Kouvaritakis, J.A: Rossiter, A.O.T. Chang: "Stable generalized predictive control: an algorithm with guaranteed stability". IEE proceedings-D, vol. 139, no. 4, pp. 349-369. July 1992.

[Lars 78] R.E. Larson, J.L. Casti: "Principles of Dynamic Programming. Part I. Basic Analytic and Computational Methods". Marcel Dekker. 1978.

[Lars 82] R.E. Larson, J.L. Casti: "Principles of Dynamic Programming. Part II. Advanced Theory and Applications". Marcel Dekker. 1982.

[Lewi 86] F.L. Lewis: "Optimal Control". John Wiley & Sons. 1986.

[Luen 79] D.G. Luenberger: "Introduction to Dynamic Systems. Theory, Models and Applications". John Wiley & Sons. 1979.

[Luqu 83] E. Luque, L. Moreno, I. Serra: "A Time-Optimal Bang-Bang control Algorithm for microprocessor Control". 1983 American Control Conference, pp. 750-751. San Francisco, California. 1983.

[Mare 90] A.J. Maren, C.T. Harston, R.M. Pap: "Handbook of Neural Computing Applications". Academic Press. 1990.

[Matl 92] "MATLAB. Reference Guide". The MathWorks. 1992

[McCl 86] J.L. McClelland, D.E. Rumelhart, PDP Research Group: "Parallel Distributed Processing. Exploration in the Microstructure of Cognition. Volume 2: Psychological and Biological Models". MIT Press. 1986.

[Mill 90] W.T. Miller, R.S. Sutton, P.J. Werbos: "Neural Networks for Control". MIT Press. 1990.

[Mins 69] M.L. Minsky, S. Papert: "Perceptrons". MIT Press. 1969.

[More 85] L. Moreno, I. Serra, E. Luque, J. Bosh: "Sub-Optimal Control Implementation Based on Microprocessor". Int. Journal of Microcomputer Applications, vol. 4, no. 3, pp. 84-87. 1985

[More 86] L. Moreno, I. Serra, E. Luque, J. Bosh: "An Extension on Sub-Optimal Control Strategy for Microprocessor Implementation". Int. Journal of Microcomputer Applications, vol. 5, no. 2, pp. 49-54. 1986

[More 92] L. Moreno, L. Acosta, J.L. Sánchez. "Design of Algorithms for the Spatial-Time Reduction Complexity of Dynamic Programming." IEE Part D. Control Theory and Applications. pp. 302-308. March 1992.

[More 93a] L. Moreno, J.D. Piñeiro, J.L. Sánchez, S. Mañas, J. Merino, L. Acosta, A. Hamilton: "Application of Neural Networks to Automated Brain Maturation Study". en "Neural Networks and Genetic Algorithms", R.F. Albretch, C.R. Reeves, N.C. Steele, (Eds.). Springer Verlag, pp. 125-130. 1993

[More 93b] L. Moreno, L. Acosta, A. Hamilton, J. Sánchez, J. Piñeiro, J. Merino "Neural Networks Based Minimum-Time Optimal Control of Nonlinear Systems" World Congress on Neural Networks, vol. 3, pp. 327-330, Portland, July 11-15, 1993.

[More 94] L. Moreno, L. Acosta, A. Hamilton, J.A. Méndez, J.L. Sánchez, J.D. Piñeiro: "Dynamic Programming Approach for Nonlinear Systems". IEE Proc.-Control Theory Appl., vol. 141, no. 6, pp. 409-417. November 1994

[More 95] L. Moreno, J.D. Piñeiro, J.L. Sánchez, S. Mañas, J. Merino, L. Acosta, A. Hamilton: "Brain Maturation Estimation Using Neural Classifier". IEEE Transactions on Biomedical Engineering. vol. 42, no. 4, pp. 428-432. April 1995.

[Nare 90] K.S. Narendra, K. Parthasarathy: "Identification and Control of Dynamic Systems Using Neural Networks." IEEE Trans. Neural Networks, vol. 1, pp. 4-27. 1990.

[Nare 91] K.S. Narendra: "Intelligent Control". IEEE Control Systems, pp. 39-40. January, 1991.

[Neur 91] NeuralWare: "Neural Computing". NeuralWare. 1991.

[Nguy 90] D.H. Nguyen, B. Widrow: "Neural Networks for Self-Learning Control Systems". IEE Control Systems Magazine, vol. 10, no. 3, pp. 18-23. April 1990.

[Pari 94] T. Parisini, R. Zoppoli : "Neural Networks for Feedback Feedforward Nonlinear Control Systems". IEEE Transactions on Neural Networks, Vol. 5, No. 3, pp. 436-449. 1994.

[Patr 89] A. Parta, G.P. Rao: "Continuous-Time approach to Self-Tuning Control: Algorithm, Implementation and Assessment". IEE Proc., vol. 136 Pt. D, no. 6. November 1989.

[Pére 94] A. Pérez, M Santos, S. Dormido, F. Morilla: "Constrained Generalized Predictive Control with Dynamic Programming". Advances in Model-Based Predictive Control, pp. 276-290. D.W. Clarke, editor. Oxford University Press. 1994.

[Pun 69] L. Pun: "Introduction to Optimization Practice". John Wiley & Sons. 1969

[RaoG 83] G.P. Rao: "Piecewise Constant Orthogonal Functions and their Application to Systems and Control." Springer Verlag. 1983.

[RaoS 84] S.S. Rao: "Optimization: Theory and Applications (Second Edition)". Wiley Eastern. 1984.

[Robi 91] B.D. Robinson, D.W. Clarke: "Robustness effects of prefilter in generalized predictive control". IEE proceedings-D, vol. 138, no. 1, pp. 2-8. January 1991.

[Rubi 86] J.E. Rubio: "Control and Optimization. The Linear Treatment of Nonlinear Problems". Manchester University Press. 1986.

[Rume 86] D.E. Rumelhart, J.L. McClelland, PDP Research Group: "Parallel Distributed Processing. Volume 1: Foundations". MIT Press. 1986.

[Ryan 82] E.P. Ryan: "Optimal Relay and Saturating Control System Synthesis". Peter Peregrinus. 1982.

[Serr 84] I. Serra, L. Moreno, E. Luque: "Time-Optimal Control Algorithm for Microprocessor with asymmetrical bounds". IEE Proc., vol. 131 Pt. D., no. 6. November, 1984.

[Shul 67] D.G. Schultz, J.L. Melsa: "State Functions and Linear Control Systems". McGraw-Hill. 1967

[Smit 88] W.A. Smith: "Análisis Numérico". Prentice-Hall Hispanoamericana. 1988.

[Snie 92] M. Sniedovich: "Dynamic Programming". Marcel Dekker. 1992.

[Sont 90] E.D. Sontag: "Mathematical Control Theory". Springer Verlag. 1990.

[Tayl 92] J.G. Taylor, C.L.T. Mannion (Eds.): "Theory and Applications of Neural Networks". Springer-Verlag. 1992.

[Wals 94] G. Walsh, D. Tibury, C. Satry, R. Murray, J.P. Laumond: "Stabilization of Trajectories for Systems with Noholonomic Constraints" IEEE Transactions on Automatic Control, vol. 39, no. 1. January 1994.

[Wass 89] P.D. Wasserman: "Neural Computing". Van Nostrand Reinhold. 1989.

[Werb 91] P.J. Werbos: "An Overview of Neural Networks for Control". IEEE Control Systems, pp. 40-41. January, 1991.

[Whit 86] P. Whittle: "Optimization Over Time. Dynamic Programming and Stochastic Control. Volume 1 & 2". John Wiley & Sons. 1986.