Arquitectura de Software Flexible y Genérica para Métodos del tipo Newton

Por

Ricardo Serrato Barrera

Tesis sometida como requisito parcial para obtener
el grado de

MAESTRO EN CIENCIAS EN LA ESPECIALIDAD DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Noviembre 2011
Tonantzintla, Puebla

Supervisada por:

Dr. Gustavo Rodríguez Gómez
Investigador del INAOE
Dr. Saúl Pomares Hernández
Investigador del INAOE

Abstract

In this thesis we present the analysis and design of an object oriented architecture that tries to give a solution to the reuse problem on Newton type methods. The proposed architecture abstracts the main characteristics of Newton type Methods using architectural and design patterns. The architecture was designed for different nonlinear problems and it is helpful as a generic interface among Newton Methods. In the thesis we describe the causes that lead to the design solutions and the problems involved when building an architecture of this type. In order to measure the extensibility and flexibility of the architecture, we use the instability and abstraction metrics. The results show that the most abstract part of the architecture is sufficiently stable to make it grow safely.

Resumen

En esta tesis se presenta el análisis y diseño de una arquitectura orientada a objetos que pretende dar solución al problema de reuso en los métodos del tipo Newton. La arquitectura propuesta abstrae las características principales en los métodos Newton por medio de patrones de arquitectura y diseño. La arquitectura está diseñada para utilizarse en diferentes problemas no lineales y servir como una interfaz genérica entre los distintos métodos tipo Newton. Durante la tesis se describen las causas que llevan a las soluciones de diseño y los problemas que conlleva la construcción de una arquitectura de este tipo. Para medir la extensibilidad y flexibilidad de la arquitectura se usaron las métricas de abstracción e inestabilidad. Los resultados mostraron que la parte más abstracta de la arquitectura es lo suficientemente estable como para hacerla crecer de forma segura.

Agradecimientos

Quiero agradecer a mis asesores Dr. Gustavo Rodríguez Gómez y Dr. Saúl Pomares Hernández por guiarme en mi trabajo de tesis. En especial al Dr. Gustavo, por mostrarme un enfoque de diseño de software que desconocía, además de propiciar en mí, un reencuentro ameno con las matemáticas.

Muchas gracias a mis sinodales Dr. Aurelio López López, Dra. María del Pilar Gómez Gil y Dr. Jesús Antonio González Bernal, por sus observaciones y comentarios.

Gracias a CONACYT por su apoyo a través de la beca No. 40306.

Dedicatorias

Para mis padres.

Para los que nunca terminan de buscar; los
que cargan sus abismos a diario.

Índice general

1. Introducción **1**

1.1. Métodos del tipo Newton 2

1.2. Problema 2

1.3. Propuesta de solución 4

1.4. Objetivo general 5

1.5. Objetivos específicos 5

1.6. Metodología 5

1.6.1. Análisis de variabilidad y partes comunes (AVC) en métodos del tipo Newton 5

1.6.2. Diseño e identificación de patrones 5

1.6.3. Implementación del prototipo 6

1.6.4. Verificación y validación 6

2. Marco Teórico **7**

2.1. Métodos del tipo Newton 7

2.1.1. Preliminares 7

2.1.2. Tipos de convergencia 8

2.1.3. Problemas no lineales 8

2.1.4. Métodos del tipo Newton para funciones de una variable . 9

2.1.5. Generalización del método 12

2.1.6. Suposiciones, ventajas y problemas de los métodos tipo Newton 13

2.1.7. Líneas de búsqueda 15

2.1.8. Regiones de confianza 17

2.1.9. Quasi-Newtons 19

2.1.10. Métodos híbridos 21

2.2. Enfoque patrón orientado a objetos 22
2.2.1. ¿Qué es un patrón? 22
2.2.2. Ejemplo: patrón adaptador 24
2.2.3. El cómo y el cuándo en los patrones 25
2.2.4. Tipos de patrones 25
2.2.5. Cómo los patrones resuelven problemas de diseño 26
2.2.6. Métricas de inestabilidad y abstracción 28
2.2.7. Análisis de variabilidad y partes comunes 30

3. Estado del arte 35
3.1. Lenguajes procedimentales, enfoque OO y problema de reuso . . . 35
3.2. Acerca del compromiso entre desempeño y flexibilidad 39
3.3. Software con métodos del tipo Newton 40
3.3.1. PETSC 41
3.3.2. COOOL 41
3.3.3. OPT++ 42

4. Análisis y diseño de la arquitectura 43
4.1. Cimientos de la arquitectura 43
4.1.1. Subsistemas y arquitectura base 43
4.1.2. Arquitectura en capas 51
4.1.3. Capa 4: Newton Genérico 51
4.1.4. Capa 3: Métodos Newton 51
4.1.5. Capa 2: Componentes Newton 52
4.1.6. Capa 1: Variaciones método Newton 52
4.2. Problemas no lineales y patrón Estado 53
4.3. Diseño Detallado 60
4.3.1. Líneas de búsqueda 60
4.3.2. Regiones de confianza 62
4.3.3. Derivadas 65
4.3.4. Creación de objetos 67
4.3.5. Interacción con paquetes externos 70
4.3.6. Evitando objetos mal cocinados 70
4.4. Resumen de patrones 71
4.4.1. Descripción de patrones 71

4.4.2. Vista general de la arquitectura y los patrones 74

5. Compromisos en el diseño 77

6. Verificación y validación 81

6.1. Caso de estudio: estimación de parámetros para un motor de corriente continua 81

6.1.1. Descripción del caso de estudio 82

6.1.2. Descripción de los métodos de optimización 83

6.1.3. Requerimientos del caso de estudio 84

6.1.4. Evaluación de los métodos de optimización 85

6.2. Midiendo el diseño 86

7. Conclusiones 93

7.1. Acerca del análisis 93

7.2. Acerca del diseño 94

7.3. Aportes 96

7.4. Trabajo futuro y posibles líneas de investigación 96

A. Distribución de las clases en los paquetes 105

A.1. Paquete *Default* 105

A.2. *Paquete algebralineal* 105

A.3. Paquete *arquitecturabase* 106

A.4. Paquete *criteriosdeparo* 106

A.5. Paquete *derivadas.* 106

A.6. Paquete *direcciones* 107

A.7. Paquete *fabricaabstracta* 108

A.8. Paquete *funciones* 108

A.9. Paquete *JacobianMatriz.Abstraccion* 109

A.10. Paquete *JacobianMatriz.principal* 109

A.11. Paquete *lineasdebusqueda* 110

A.12. Paquete *motor* 111

A.13. Paquete *newtonsconcretos* 111

A.14. Paquete *PrecisionNumerica* 112

A.15. Paquete *problemasnolineales* 112

A.16. Paquete *RegionesDeConfianza* 112

A.17. Paquete *vistas* 113

B. Abstracción e inestabilidad en cada paquete 115

B.1. Arquitectura base y cuatro paquetes principales 116

B.2. Paquetes con el diseño de líneas de búsqueda incorporado 117

B.3. Paquetes con el diseño de líneas de búsqueda y regiones de confianza 118

B.4. Todos los paquetes que conforman la arquitectura (hasta la escritura de esta tesis) 119

C. Resumen de notación UML 121

Índice de figuras

2.1. Método de Newton para la función g(x)=x^2. 10
2.2. Método Newton para optimización sin restricciones. 11
2.3. Método de la secante. 20
2.4. Funcionamiento de los métodos de "pata de perro". 22
2.5. Patrón adaptador. 25
2.6. Zonas ideales en donde situar los paquetes [Mar03]. 30
2.7. Zonas en donde pueden estar los paquetes [Mar03]. 31

4.1. Dependencia entre cada paso del algoritmo genérico y cada componente. 46
4.2. Patrón Fachada. 47
4.3. Patrón Plantilla. 48
4.4. Patrón Puente. 50
4.5. Grados de abstracción de los patrones. 50
4.6. Arquitectura base. 53
4.7. Transición de problemas no lineales en un gradiente descendente. ∇f es el gradiente de f (SNL: sistema no lineal. MC: mínimos cuadrados). 55
4.8. Posible transición entre optimización sin restricciones (OSR) y mínimos cuadrados (MC). Hf es la matriz hessiana de f. 56
4.9. Cálculo de direcciones en el método híbrido de Powell. J es la matriz jacobiana y s es la dirección a encontrar. 57
4.10. Transición entre problemas para línea de búsqueda. 58
4.11. Transición de estados para calcular la dirección Newton en regiones de confianza. 58
4.12. Patrón Estado. 59

4.13. Patrón Estado para problemas no lineales. 60
4.14. Patrón Estrategia. 62
4.15. Diseño de líneas de búsqueda. 63
4.16. Diseño para la solución del problema con restricciones y concordancia entre modelo y función. 64
4.17. Diseño para regiones de confianza. 65
4.18. Variaciones en el cálculo de las derivadas. 66
4.19. Interacción entre las derivadas y el patrón Estado. 66
4.20. Patrón Fábrica Abstracta. 68
4.21. Patrón Singleton. 69
4.22. Diseño para la creación de objetos. 70
4.23. Diseño para interactuar con paquetes de externos. 71
4.24. Creación de objetos y asignación de referencias entre los mismos. . 72
4.25. Vista general de la arquitectura y ubicación de los patrones. . . . 75

6.1. Convergencia de los métodos usando línea de búsqueda. 87
6.2. Convergencia de los métodos sin usar línea de búsqueda. 87
6.3. Inestabilidad y abstracción de la arquitectura base y cuatro paquetes principales. Notará que los paquetes *direccion, funciones, criteriosparo* y *lineasbusqueda* están traslapados en (1,0). Puede consultar una descripción detallada de estos resultados en el apéndice B. 90
6.4. Inestabilidad y abstracción de los paquetes tomando en cuenta el diseño de las líneas de búsqueda. 90
6.5. Inestabilidad y abstracción de los paquetes al agregar el diseño de lineas de búsqueda y regiones de confianza. 91
6.6. Inestabilidad y abstracción de los paquetes tomando en cuenta líneas de búsqueda, regiones de confianza y el caso de estudio. . . 92

B.1. Inestabilidad y abstracción de la arquitectura base y cuatro paquetes principales. 116
B.2. Inestabilidad y abstracción de los paquetes tomando en cuenta el diseño de las líneas de búsqueda. 117
B.3. Inestabilidad y abstracción de los paquetes al agregar el diseño de lineas de búsqueda y regiones de confianza. 118

B.4. Inestabilidad y abstracción de todos los paquetes que conforman la arquitectura hasta la escritura de esta tesis. 120

Índice de Tablas

2.1. Método Newton con la función f(x) =arctan x 15
2.2. Forma de organizar variaciones en la matriz de análisis [ST01] . . 33
2.3. Matriz de análisis para agentes que juegan futbol 33
2.4. Matriz de análisis expandida para agentes que juegan futbol . . . 33
2.5. Matriz de análisis para el concepto Matriz 34

3.1. Diseño OO y patrones en cómputo científico 39
3.2. Software para métodos Newton 41

4.1. Análisis de variabilidad y partes comunes sobre los conceptos dirección y longitud de paso 45
4.2. Matriz de análisis problemas no lineales. (SNL: sistema no lineal. OSR: optimización sin restricciones. MC: mínimos cuadrados) . . 55
4.3. Conceptos y sus variaciones en líneas de búsqueda 61
4.4. Conceptos y variaciones en regiones de confianza 63

5.1. Compromisos de diseño presentes en la arquitectura 80

6.1. Parámetros usados en el caso de estudio. 83
6.2. Error absoluto y relativo de los métodos usando línea de búsqueda 88
6.3. Error absoluto y relativo de los métodos sin usar línea de búsqueda 88

B.1. Inestabilidad y abstracción de la arquitectura base y cuatro paquetes principales 116
B.2. Inestabilidad y abstracción de los paquetes tomando en cuenta el diseño de líneas de búsqueda. 117
B.3. Inestabilidad y abstracción de los paquetes tomando en cuenta el diseño de líneas de búsqueda y regiones de confianza 118

B.4. Inestabilidad y abstracción de todos los paquetes que conforman la arquitectura hasta la escritura de esta tesis 119

Capítulo 1

Introducción

El área de cómputo científico se encarga del estudio y desarrollo de técnicas numéricas para la solución de modelos matemáticos que representan fenómenos físicos [GO92]. Se puede decir que es un área en donde intervienen las matemáticas y las ciencias computacionales con la finalidad de resolver problemas científicos y de ingeniería.

Los estudios llevados a cabo en el área de cómputo científico giran en torno al modelo matemático. "*Un modelo matemático es una construcción matemática abstracta, y simplificada, de una parte de la realidad, que es creada con un propósito en particular*" [Ben00]. El modelo imita ciertas características, relevantes, en un objeto de estudio y puede llegar a ser simple o muy complejo. Por ejemplo, el modelo puede caracterizar el cambio climático en una determinada zona geográfica, puede tratar de predecir el cambio demográfico en un país, simular el comportamiento de una planta eléctrica, o bien, imitar la locomoción de un cuadrúpedo.

Un modelo suele ser expresado mediante un sistema de ecuaciones algebraico-diferenciales, y dependiendo de la finalidad del mismo, involucra la solución de distintos problemas. En esta tesis nos interesan tres problemas principalmente: i) la resolución del modelo cuando puede verse como un sistema de ecuaciones no lineal, ii) el problema de optimización, en donde el modelo es una función a la que hay que minimizar, iii) el problema de mínimos cuadrados en donde se cuenta con un conjunto de datos y se busca ajustar los coeficientes del modelo para que concuerde con los datos.

1.1. Métodos del tipo Newton

Para resolver los tres problemas mencionados anteriormente se suele recurrir a métodos numéricos y algunos de los más utilizados son los métodos del tipo Newton. A grosso modo, el método Newton se encarga de encontrar un punto en donde una función decrezca con respecto a la iteración previa o se acerque a la solución del sistema no lineal. La cualidad principal del método en su forma original, es que dado un buen punto de inicio tiene convergencia q-cuadrática; es decir, el método encuentra una solución aproximada, durante cada iteración, que se acerca a la solución de forma exponencial.

Desafortunadamente, si el punto inicial está alejado de la solución el desempeño del método es pobre. A pesar de esta desventaja, las cualidades de convergencia del método son deseables. Por esta razón se ha hecho un estudio extenso en torno al método y se han propuesto diferentes variaciones del mismo. Por ejemplo, se han desarrollado técnicas, como las llamadas líneas de búsqueda y regiones de confianza [WhN99], que propician un buen comportamiento cuando el punto inicial está lejos de la solución. A su vez los Quasi-Newtons [XZ01] y los Newton inexactos o truncados [Nas00], reducen su costo computacional sacrificando precisión. También se ha tratado de combinar las virtudes del método Newton con otros métodos dando como resultado los llamados métodos híbridos [Blu80, Pow70]. Otras ideas, como los métodos tensoriales [BS97], intentan reducir el error dado por la aproximación a la función de evaluación en el método Newton original.

Se puede decir que la teoría asociada a los métodos del tipo Newton es vasta y madura, lo que conlleva a que exista una diversidad considerable de métodos de este tipo, teniendo cada uno, una serie de cualidades y características que pueden ser aprovechadas para resolver eficaz y eficientemente un problema dado.

1.2. Problema

La mayor parte del software para cómputo científico, incluyendo lo concerniente al método Newton, está escrito en lenguajes procedimentales cuyas desventajas se mencionan a continuación:

- Dificultad para adaptar la interfaz de las paqueterías de software con la aplicación deseada, e incompatibilidad con estructuras de datos [GPS99,

DLh+94].

- Necesidad de aprender a usar cada paquetería e implementar una interfaz específica para cada una [MOHW07].
- Implementación de prototipos de nuevos algoritmos es lenta ya que, en general, se hace desde cero [MCH+04].
- Falta de naturalidad para expresar los algoritmos que hay detrás del código [GPS99].

En la actualidad existen paqueterías de software con algoritmos de métodos del tipo Newton que han mostrado ser eficientes y robustas. No obstante, las paqueterías escritas en lenguajes procedimentales como MINPACK[MGH80], HOMPACK [WBM87] y MINOS [MS83] presentan las dificultades ya mencionadas. Otra problemática es que la mayoría de métodos Newton están dispersos en las diversas paqueterías lo que hace díficil buscar el adecuado para un problema en específico. Asimismo, algunas paqueterías centran su atención en los métodos Newton espécificos para su área, tal es el caso OPT++ [MOHW07] en optimización y PETSC [BGMS95] en ecuaciones diferenciales; es decir, no hacen un intento para diseñar software con métodos Newton para propósitos generales.

La selección del método Newton para un problema dado puede ir desde algo trivial hasta algo complejo si la aplicación así lo amerita, e incluso se puede requerir de modificaciones en el método seleccionado con el fin de mejorarlo. Desde la perspectiva de un usuario inexperto, lo más sencillo es tomar distintos métodos Newton y probar cuál es el que brinda mejores resultados. El caso de un usuario experimentado puede ser más complejo. El experto puede optar por un método con base en su conocimiento y experiencia, luego puede modificar el método seleccionado con la intención de mejorarlo. En los dos casos, el software actual no ofrece las cualidades necesarias para facilitar los cambios o el intercambio entre los distintos métodos Newton.

El enfoque orientado a objetos representa a los elementos del sistema como objetos débilmente acoplados [Som04] lo que permite crear software con las cualidades de reuso, flexibilidad, extensibilidad y fácil mantenimiento. Lamentablemente, este enfoque no se ha difundido ampliamente en el área de cómputo científico debido a que introduce un costo de cómputo extra que repercute en el desempeño del software, creando así un compromiso entre flexibilidad y desempeño que

muchas veces se prefiere no tomar [Bli02]. A pesar de esto, en la actualidad existe software orientado a objetos que presenta un buen desempeño en sus aplicaciones correspondientes [MCH+04, DLh+94, MMG+00, GPS99]. Aunado a lo anterior, la innovación tecnológica ha brindado procesadores más rápidos, plataformas multinúcleo y capacidad de paralelizar software, lo que hace factible romper dicho compromiso.

1.3. Propuesta de solución

En esta tesis se plantea el uso del enfoque orientado a objetos como una solución al problema de reuso que existe en los métodos del tipo Newton y se enfoca en la fase de diseño de software. El diseño orientado a objetos involucra un estudio detallado del problema para encontrar los objetos, las relaciones entre ellos, la granularidad y la abstracción adecuada [GHJV95]. La etapa de diseño es un proceso creativo en donde los diseñadores tienen que recurrir a su experiencia, y a la de los demás, para realizar buen software [GHJV95]. El inconveniente es que la experiencia en el diseño de software orientado a objetos continúa inmadura en el área de cómputo científico. Si bien existe software orientado a objetos, no existen muchos trabajos en donde se detallen las ideas de diseño que hay detrás del software construido y menos aún en lo que concierne a los métodos Newton.

El reuso de software en cómputo científico se ha centrado en la reutilización de librerías escritas principalmente en lenguajes procedimentales. Otras formas de reuso que no han sido debidamente explotadas son el reuso de diseño de software, arquitecturas de dominio específico y patrones de software. Los patrones de software son buenas soluciones a problemas de diseño que se presentan de forma recurrente [GHJV95], mientras que las arquitecturas de dominio específico abstraen las características esenciales de diversos sistemas en un dominio [Som04]. Una arquitectura es lo suficientemente genérica para servir como los cimientos de diversas construcciones de software y representa una buena forma de reuso ya que en ella se plasma el cómo deben interactuar los diferentes elementos que conforman a un sistema [FK05].

Debido a la poca experiencia en diseño de software orientado a objetos en el área de cómputo científico, en esta tesis se retoma la experiencia de diseñadores de software administrativo, o de negocios, en forma de patrones software [GHJV95,

BMR+08, ST01]. Esto con el objeto de construir una arquitectura que abstraiga las principales características de los métodos del tipo Newton.

1.4. Objetivo general

El objetivo principal de esta tesis es diseñar una arquitectura de software patrón orientada objetos, flexible y genérica, para métodos del tipo Newton.

1.5. Objetivos específicos

- Realizar un análisis de los distintos métodos del tipo Newton.
- Identificar y evaluar los patrones a utilizar en la arquitectura.
- Diseñar la arquitectura.

1.6. Metodología

1.6.1. Análisis de variabilidad y partes comunes (AVC) en métodos del tipo Newton

El análisis de variabilidad y partes comunes (AVC) es propuesto por Coplien [CHW02] y consiste en identificar conceptos del problema, para luego encontrar sus variaciones en distintos escenarios. En este trabajo, dicho análisis se lleva a cabo sobre los siguientes métodos:

- Métodos Newton que usan líneas de búsqueda
- Métodos Newton que usan regiones de confianza
- Quasi-Newtons
- Métodos híbridos tipo pata de perro o *dogleg.*

El resultado del análisis es sintetizado en una matriz de análisis [ST01] que sirve para visualizar las posibles clases y los patrones que puedan ser aplicados.

1.6.2. Diseño e identificación de patrones

Durante esta fase se identifican los problemas de diseño que se presentan durante el diseño de la arquitectura. Luego se utiliza la clasificación de patrones de Gamma [GHJV95] y el resultado del AVC para encontrar los posibles patrones que resuelvan el problema de diseño tratado.

La fase de diseño de la arquitectura se divide en dos partes:

- Diseño arquitectónico: aquí se encuentran los elementos de diseño o componentes que conforman el sistema. Éstos se representan como paquetes de software. En esta fase se construye una arquitectura base que abstrae las características comunes entre los diversos métodos Newton y se definen interfaces hacia las diferentes variantes en el método Newton.

- Diseño detallado: aquí se tratan los aspectos del diseño que son propios de cada método tipo Newton. Es decir, se tratan los aspectos en los que varía cada método Newton.

1.6.3. Implementación del prototipo

Esta tesis se centra en la fase de diseño. No obstante, para evaluar el impacto del diseño en el desarrollo de software se implementó la arquitectura y se extendió para incorporar algunos métodos Newton.

1.6.4. Verificación y validación

En la verificación se muestra que el diseño fue construido de forma correcta por medio de las métricas de inestabilidad y abstracción de Robert Martin [Mar03]. Para la validación se usa un caso de estudio en donde el diseño tiene que cumplir con los requerimientos de acuerdo a las necesidades de un posible usuario en ese caso de estudio.

Capítulo 2

Marco Teórico

En este capítulo se cubre la teoría necesaria para entender el diseño de la arquitectura propuesta. En la primera parte del capítulo se tocan aspectos básicos acerca de los métodos del tipo Newton, una descripción detallada se puede encontrar en [Kel99, Kel03, DS96, WhN99]. En la segunda sección de este capítulo se explica el enfoque patrón orientado a objetos.

2.1. Métodos del tipo Newton

2.1.1. Preliminares

A lo largo de este capítulo se denotan a los vectores con negritas y a los escalares con letras normales. Se tratan funciones con dominio en $\mathbb{R}^n$ y rango en $\mathbb{R}^m$, por ejemplo:

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{pmatrix},$$

en donde cada i-ésima función componente f_i tiene dominio en $\mathbb{R}^n$ y rango en $\mathbb{R}$, $\|\cdot\|$ denota a la norma euclidiana en $\mathbb{R}^n$:

$$\|\mathbf{x}\| = \left(\sum_{i=1}^{n} x_i^2\right)^{1/2}.$$

2.1.2. Tipos de convergencia

Se dice que una sucesión $\{\mathbf{x}_n\} = \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2 \ldots$ *converge* a $\mathbf{x}_*$ si cumple con la expresión (2.1) [DS96].

$$\lim_{n\to\infty} \|\mathbf{x}_n - \mathbf{x}_*\| = 0 \tag{2.1}$$

La sucesión tiene convergencia *q-superlineal* cuando cumple con la propiedad (2.2) y converge de forma *q-cuadrática* si existe una constante $c > 0$ tal que cumpla con la expresión (2.3) cuando $q = 2$. Si $c \in (0, 1)$ y $q = 1$ la secuencia converge de forma *q-lineal*[DS96, Kel99].

$$\lim_{n\to\infty} \frac{\|\mathbf{x}_{n+1} - \mathbf{x}_*\|}{\|\mathbf{x}_n - \mathbf{x}_*\|} = 0 \tag{2.2}$$

$$\|\mathbf{x}_{n+1} - \mathbf{x}_*\| \leq c \|\mathbf{x}_n - \mathbf{x}_*\|^q \tag{2.3}$$

2.1.3. Problemas no lineales

A continuación se describen los problemas que pueden ser tratados con métodos del tipo Newton. Primero, se define una función lineal como una función que cumple con las características (2.4) y (2.5) [GOS92]. Entenderemos por *función no lineal* a una función que no cumpla con alguna de las características mencionadas.

$$\mathbf{F}(\mathbf{u} + \mathbf{v}) = \mathbf{F}(\mathbf{u}) + \mathbf{F}(\mathbf{v}) \tag{2.4}$$

$$\mathbf{F}(\alpha\mathbf{u}) = \alpha\mathbf{F}(\mathbf{u}). \tag{2.5}$$

Un problema no lineal involucra tratar con funciones no lineales. Los problemas no lineales que se pueden resolver por medio de métodos Newton se detallan a continuación [DS96]:

- **Sistemas de ecuaciones no lineales**

  Dada una función $\mathbf{F} : D \subset \mathbb{R}^n \to \mathbb{R}^n$, encontrar una $\mathbf{x}_* \in D$ que $\mathbf{F}(\mathbf{x}_*) = 0$.

- **Optimización sin restricciones**

  Dada una función $f : D \subset \mathbb{R}^n \to \mathbb{R}$, encontrar una $\mathbf{x}_* \in D$ que $f(\mathbf{x}_*) \leq f(\mathbf{x})$ para cada $\mathbf{x} \in D$.

- **Mínimos cuadrados**

  Dada una función $\mathbf{G} : D \subset \mathbb{R}^n \to \mathbb{R}^m$, $m \geq n$, encontrar una $\mathbf{x}_* \in D$ que minimice a $\sum_{i=1}^{m} (\mathbf{r}_i(\mathbf{x}))^2$, donde $\mathbf{r}_i$ es la i-ésima función componente de $\mathbf{G}$.

2.1.4. Métodos del tipo Newton para funciones de una variable

Con la intención de ilustrar el comportamiento de los métodos del tipo Newton, se mostrarán algunos ejemplos sencillos usando funciones de una variable. Una descripción más detallada de lo tratado en esta sección puede encontrarse en [DS96].

Los métodos del tipo Newton son métodos iterativos que tienen la secuencia

$$x_{n+1} = x_n + \lambda s,$$

donde a λ se le llama longitud de paso y s es la dirección Newton. Tanto λ como s se calculan con la intención de resolver de manera progresiva a alguno de los problemas mencionados en la sección anterior. Por ahora, nos centraremos en saber cómo se calcula la dirección Newton, los detalles acerca del cálculo de la longitud de paso se darán mas tarde.

Partiremos por recordar la serie de Taylor (2.6) [GOS92]; mientras más se expanda la serie se logrará una mejor aproximación de la función dada y si se expandiera hasta el infinito el polinomio resultante sería igual a dicha función.

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n \tag{2.6}$$

Si la serie se trunca hasta la primera derivada se obtiene una aproximación, o modelo lineal (2.7) a la función en el punto $(a, f(a))$. La expansión hasta la segunda derivada se le conoce como modelo cuadrático (2.8). Para una función de una variable, el modelo lineal y cuadrático son una línea y una parábola, respectivamente, que pasan por el punto $(a, f(a))$.

$$m(x) = f(a) + f'(a)(x-a) \tag{2.7}$$

$$M(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 \tag{2.8}$$

Considere el problema de encontrar la solución de la ecuación $x^2 = 0$, con una estimación inicial $x_0 = 1$ y se quiere hacer por medio de aproximaciones sucesivas usando el modelo lineal. Hay que notar que es el caso más simple al resolver un sistema de ecuaciones ya que sólo está formado por una variable y una ecuación. Lo que se pretende es resolver el modelo lineal (2.7) haciendo $m(x) = 0$ y usando $a = x_0$ para así obtener una x_1 más cerca de la solución (ver figura 2.1). Al igualar el modelo a cero y despejar x se obtiene

$$x = a - \frac{f(a)}{f'(a)},$$

que puede ser visto como la siguiente secuencia

$$x_{n+1} = x_n + s, \text{ con } s = -\frac{f(x_n)}{f'(x_n)},$$

que es el método Newton original para funciones de una variable o también llamado Newton-Raphson. El comportamiento del método para la función $g(x) = x^2$ está en la figura 2.1.

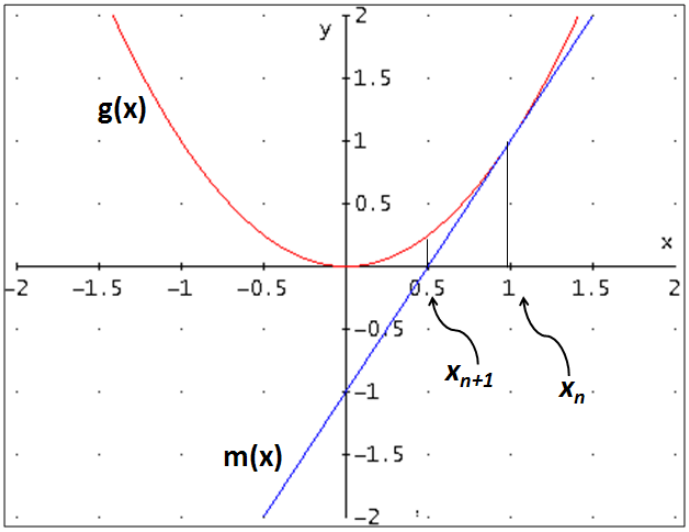


Figura 2.1: Método de Newton para la función g(x)=x^2.

El método Newton-Raphson sirve para encontrar la solución de una ecuación

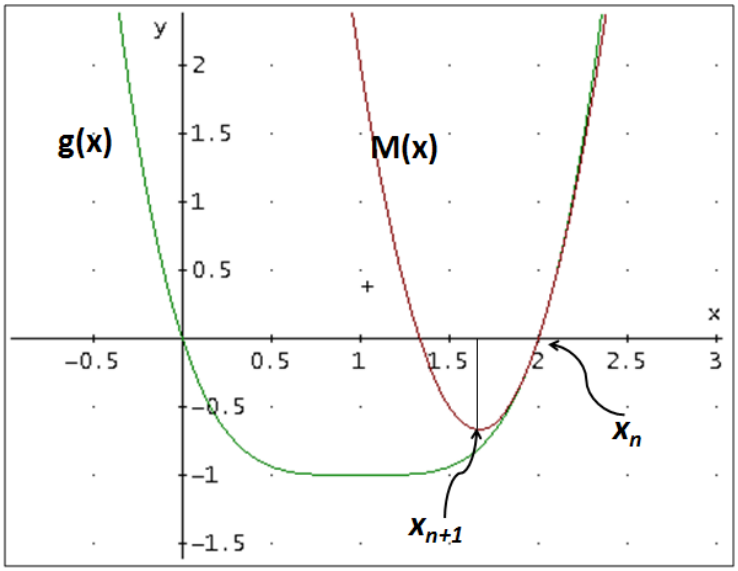


Figura 2.2: Método Newton para optimización sin restricciones.

porque resuelve el modelo lineal cuando $m(x) = 0$. Sin embargo, esto no sirve bien para un problema de optimización sin restricciones. Considere ahora la función $g(x) = (x-1)^4 - 1$, el valor de x que minimiza a $g(x)$ es uno (ver figura 2.2). Note que usando el método anterior la solución converge hacia un punto en que $g(x) = 0$ y no al minimizador real. Usando el modelo cuadrático (2.8) se puede llegar a una aproximación del minimizador, encontrando el punto estacionario del modelo, es decir, cuando $M'(x) = 0$. Derivando el modelo cuadrático (2.8) e igualando a cero se obtiene

$$f'(a) + f(a)''(x-a) = 0,$$

$$x = a - \frac{f'(a)}{f''(a)},$$

que puede ser vista como la secuencia

$$x_{n+1} = x_n + s, \text{ con } s = -\frac{f'(x_n)}{f''(x_n)} \tag{2.9}$$

y que es el método Newton para problemas de optimización sin restricciones y funciones de una variable.

El problema de mínimos cuadrados se puede ver como un caso especial del

problema de optimización sin restricciones. Mínimos cuadrados consiste en ajustar los coeficientes de una función con respecto a un conjunto de datos dado. Imagine una función $g(x) = ax$ y un conjunto de datos $(x_1, t_1), (x_2, t_2), ...(x_n, t_n)$, entonces el problema consistiría en encontrar el valor de a que minimice a la expresión $(t_1 - ax_1)^{\ 2} + (t_2 - ax_2)^2 ... + (t_n - ax_n)^{\ 2}$, que se trata de una función de una variable y se puede resolver con la secuencia (2.9).

2.1.5. Generalización del método

Para generalizar el método a funciones de varias variables, basta con recurrir a la serie de Taylor generalizada y formar el modelo lineal o cuadrático. Por ejemplo, si se quisiera resolver un sistema de ecuaciones no lineales el sistema estaría expresado como una función $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ y el modelo lineal sería como sigue:

$$\mathbf{m}(\mathbf{x}) = \mathbf{F}(\mathbf{x}_n) + \mathbf{JF}(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n),$$

donde $\mathbf{JF}(\mathbf{x})$ es la matriz jacobiana $\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$.

De la misma manera que en una función de una variable, la dirección Newton $\mathbf{s}$ se obtiene al hacer $\mathbf{m}(\mathbf{x}) = 0$, por lo tanto la dirección será

$$\mathbf{s} = -\mathbf{F}(\mathbf{x}_n)\mathbf{JF}(\mathbf{x}_n)^{-1}. \tag{2.10}$$

En el problema de optimización sin restricciones la función es de la forma $f : \mathbb{R}^n \to \mathbb{R}$ y el modelo cuadrático es el siguiente

$$M(\mathbf{x}) = f(\mathbf{x}_n) + \boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n)^T(\mathbf{x} - \mathbf{x}_n) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_n)^T\boldsymbol{\nabla}^2\mathbf{f}(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n),$$

donde $\boldsymbol{\nabla}\mathbf{f}(\mathbf{x})$ es el gradiente$\begin{pmatrix} \frac{\partial f(x)}{\partial x_1} & \cdots & \frac{\partial f(x)}{\partial x_n} \end{pmatrix}^T$, y

$\boldsymbol{\nabla}^2\mathbf{f}(\mathbf{x})$ es la matriz hessiana $\begin{pmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{pmatrix}$.

La dirección Newton para optimización sin restricciones es la siguiente [WhN99]:

$$\mathbf{s} = -\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n)\mathbf{Hf}(\mathbf{x}_n)^{-1}, \tag{2.11}$$

donde $\mathbf{Hf}$ es una aproximación a $\boldsymbol{\nabla}^2\mathbf{f}$ o una matriz calculada con una fórmula quasi-Newton. Cuando $\mathbf{Hf}$ es una matriz identidad el método se convierte en un gradiente descendente.

En un problema de mínimos cuadrados la función original es de la forma $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$, pero se pretende encontrar el minimizador de la función

$$\mathbf{f} = \frac{1}{2}\mathbf{F}^T\mathbf{F} = \|\mathbf{F}\|_2^2 .$$

Se sabe que [DS96, MBT04]:

$$\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T\mathbf{F} \quad \text{y}$$

$$\boldsymbol{\nabla}^2\mathbf{f}(\mathbf{x}) = \mathbf{J}(\mathbf{x})^T\mathbf{J}(\mathbf{x}) + \sum_{i=1}^{m} f_i\boldsymbol{\nabla}^2\mathbf{f}_i,$$

donde f_i es la i-ésima función componente de $\mathbf{F}$ y $\boldsymbol{\nabla}^2\mathbf{f}_i$ es su respectiva matriz hessiana. Dado lo anterior, el modelo cuadrático y la dirección Newton para optimización sin restricciones también se pueden usar para un problema de mínimos cuadrados.

2.1.6. Suposiciones, ventajas y problemas de los métodos tipo Newton

Suposiciones para sistemas de ecuaciones no lineales

Las siguientes son las suposiciones del método cuando se usa para resolver sistemas de ecuaciones no lineales [Kel03]:

- El sistema tiene solución.
- La función $\mathbf{F}$ debe ser continua y diferenciable.
- $\mathbf{JF}$ tiene que ser Lipschitz continua en Ω. Es decir, existe una $\gamma > 0$ que

$$\|\mathbf{JF}(\mathbf{x}) - \mathbf{JF}(\mathbf{y})\| \leq \gamma \|\mathbf{x} - \mathbf{y}\| , \text{ para toda } \mathbf{x},\mathbf{y} \in \Omega.$$

- $\mathbf{JF}(\mathbf{x}_*)$ no es singular.

Suposiciones para problemas de optimización

Las siguientes son las suposiciones del método cuando se usa para resolver problemas de optimización [Kel99, WhN99]:

- La función f es continua y dos veces diferenciable.
- $\boldsymbol{\nabla}^2\mathbf{f}(\mathbf{x})$ tiene que ser Lipschitz continua en Ω. Es decir, existe una $\gamma > 0$ que
$$\left\|\boldsymbol{\nabla}^2\mathbf{f}(\mathbf{x}) - \boldsymbol{\nabla}^2\mathbf{f}(\mathbf{y})\right\| \leq \gamma \left\|\mathbf{x} - \mathbf{y}\right\|, \text{ para toda } \mathbf{x},\mathbf{y} \in \Omega.$$
- $\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_*) = 0$.
- $\boldsymbol{\nabla}^2\mathbf{f}(\mathbf{x}_*)$ es definida positiva.

Ventajas

Si las suposiciones anteriores se mantienen y si se proporciona un punto suficientemente cerca a la solución del problema no lineal, el método Newton tiene convergencia q-cuadrática, lo que hace que se aproxime rápidamente a un mínimo local o a la solución del sistema no lineal. Esta característica es sumamente deseable ya que el Newton puede converger a la solución en unas cuantas iteraciones. Otra ventaja es que la mayoría de los aspectos que propician un comportamiento irregular en el método ya han sido tratados de diversas maneras.

A continuación se comenta acerca de algunos de los problemas principales en los métodos Newton y algunas de las soluciones a estos problemas.

Problemas

El Newton original puede tener un comportamiento oscilatorio si el punto inicial no está cerca de la solución. Para ilustrar esto se tomará el ejemplo de Kelley [Kel03], en donde se trata de encontrar la solución para la función $f(x) = \arctan x$, con un valor inicial $x_0 = 10$, los resultados están en la tabla (2.1). Como se puede ver, un mal punto de inicio puede provocar que el método Newton no converja hacia la solución. Por esta característica, se dice que el Newton es *localmente convergente.* Algunas de las técnicas más usadas para lograr *convergencia global*

(es decir, el Newton converge desde cualquier punto de inicio) son las líneas de búsqueda y regiones de confianza que se detallarán más adelante.

Otra desventaja del método de Newton es el cálculo de la matriz hessiana y jacobiana, en el modelo lineal y cuadrático, respectivamente. Para aplicaciones prácticas el cálculo analítico de las matrices mencionadas no es una opción. Se pueden usar técnicas de diferencias finitas que aproximan las derivadas que componen a cada matriz. El problema es que si la aplicación deseada es bastante exigente (digamos algunos cientos de ecuaciones y restricciones de tiempo real) el costo del cálculo con diferencias finitas, durante cada iteración, es demasiado. Otra opción es usar quasi-Newtons que disminuyen el tiempo computacional al usar información de iteraciones previas para hacer un estimado de las matrices en la iteración actual; sin embargo, al hacer lo anterior se sacrifica convergencia.

Tabla 2.1: Método Newton con la función f(x) =arctan x

Iteración	$f(x) = \arctan x$
1	10
2	-138
3	2.9 x 10
4	-1.5 x 10^9
5	9.9 x 10^{17}

Para obtener alguna de las direcciones Newton (2.10,2.11) se debe resolver un sistema de la forma $\mathbf{Ax} = \mathbf{b}$ que implica que $\mathbf{A}$ debe ser no singular (su determinante es diferente de cero). Cuando $\mathbf{A}$ se aproxima a la singularidad la dirección Newton puede desviarse. Algunos de los métodos más robustos ante esta situación son los métodos híbridos que tratan de hacer un cambio entre el método del gradiente descendente y el método Newton.

2.1.7. Líneas de búsqueda

Anteriormente se explicó como surge la secuencia de los métodos del tipo Newton para funciones de una variable y varias variables. También se mostró cómo calcular el paso Newton por medio del modelo lineal y cuadrático. En esta sección se toca el tema de cómo calcular la longitud del paso y es un pequeño

resumen del capítulo de líneas de búsqueda del libro de Nocedal [WhN99]. Para esto se usará una función de la forma $f : \mathbb{R}^n \to \mathbb{R}$, sin embargo, lo dicho aquí es análogo para las diferentes funciones en distintos problemas no lineales tratados en esta tesis.

Uno de los problemas principales del método Newton en su forma más simple, es que necesita de un punto de inicio cerca de la solución. Por esta razón se han desarrollado técnicas como las líneas de búsqueda que encuentran una longitud de paso que garantice el decremento de la función en cada iteración.

Un mal punto de inicio puede provocar que el método Newton no converja hacia la solución. Sin embargo, esto no quiere decir que el método sea incorrecto. La dirección Newton apunta hacia una dirección de descenso pero la magnitud de la dirección puede ser muy grande. Ante esta situación, se opta por hacer más chico el paso multiplicándolo por una longitud en el intervalo $(0, 1]$.

El problema ahora consiste en encontrar una longitud de paso λ que decrezca "mejor" la función en la siguiente iteración. Lo ideal sería encontrar el mínimo global de la función $\phi(\lambda) = f(\mathbf{x}_n + \lambda \mathbf{s}_n)$. Pero, esto puede ser caro. Por lo regular se suelen usar estrategias que encuentren una longitud que decrezca a la función en un tiempo aceptable. Primero, definiendo un intervalo en donde se encuentre un conjunto con longitudes apropiadas, y segundo, buscando la longitud en ese intervalo.

Algunos de los intervalos más usados son la condición de Wolfe, la condición de curvatura y la condición de Goldstein.

La condición de Wolfe (2.12) consiste en acotar con una línea

$$l(\lambda) = f(\mathbf{x}_n) + c_1 \lambda \boldsymbol{\nabla} \mathbf{f}(\mathbf{x}_n)^T \mathbf{s}_n$$

a las posibles longitudes de paso.

$$f(\mathbf{x}_n + \lambda \mathbf{s}_n) \leq f(\mathbf{x}_n) + c_1 \lambda \boldsymbol{\nabla} \mathbf{f}(\mathbf{x}_n)^T \mathbf{s}_n, \tag{2.12}$$

donde c_1 es una constante que cambia la pendiente de la línea $l(\lambda)$. Una c_1 muy grande es demasiado restrictiva, mientras que una pequeña permite un intervalo con más longitudes posibles.

La condición de curvatura (2.13) consiste en restringir los valores de la pendi-

ente de $\phi(\lambda)$ para los posibles valores de λ.

$$\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n + \lambda\mathbf{s}_n)^T\mathbf{s}_n \geq c_2\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n)^T\mathbf{s}_n, \tag{2.13}$$

donde c_2 es una constante que hace una función similar a c_1.

La condición de Goldstein (2.14) sigue una idea parecida a la condición de Wolfe, pero se utilizan dos líneas para acotar los valores de λ.

$$f(\mathbf{x}_n) + (1 - c_3)\lambda\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n)^T\mathbf{s}_n \leq f(\mathbf{x}_n + \lambda_n\mathbf{s}_n) \leq f(\mathbf{x}_n) + c_3\lambda\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n)^T\mathbf{s}_n \tag{2.14}$$

Las condiciones de curvatura, de Wolfe, y Goldstein sirven para saber si una longitud brinda un "buen" decremento para la función dada. Para encontrar las longitudes se puede proceder de distintas maneras. La forma más común consiste en utilizar una técnica de bisección, usando la sucesión $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, ...$ como los valores de las longitudes. Otras formas consisten en utilizar interpolación cuadrática o cúbica. Un algoritmo general para líneas de búsqueda es el siguiente.

Algoritmo 1 *Algoritmo general para líneas de búsqueda*

$s \leftarrow$ *dirección Newton*
repeat
  $\lambda \leftarrow$ *calcular por medio de bisección, interpolación cuadrática o cúbica*
until *La condición de decremento se cumpla con* $f(\mathbf{x}_n + \lambda\mathbf{s})$

2.1.8. Regiones de confianza

El funcionamiento básico de los algoritmos de las líneas de búsqueda es encontrar la dirección Newton y luego encontrar una longitud de paso apropiada para garantizar el decremento de la función. Las regiones de confianza también garantizan el decremento de la función pero lo hacen de una manera distinta. Con esta técnica se define una región con radio Δ dentro de la cual se puede confiar en la aproximación de la función por medio del modelo cuadrático o lineal. Es decir, el modelo es una "buena" aproximación de la función en los puntos que estén dentro de la región de confianza. Encontrar el minimizador del modelo cuadrático dentro de la región de confianza se puede ver como un problema de optimización

con restricciones. Haciendo $\mathbf{s} = \mathbf{x} - \mathbf{x}_n$, se tiene

$$\min_{s\in\mathbb{R}^n} M(\mathbf{x}_n + \mathbf{s}) = f(\mathbf{x}_n) + \boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n)^T\mathbf{s} + \frac{1}{2}\mathbf{s}^T\mathbf{H}\mathbf{f}(\mathbf{x}_n)\mathbf{s} \qquad sujeto\ a\ \ \|\mathbf{s}\| \leq \Delta. \tag{2.15}$$

La idea de este tipo de técnicas es hacer más grande el radio de confianza si el modelo es una buena aproximación de la función de evaluación. Por el contrario, si el modelo no es una buena aproximación, la región se hace más pequeña. Como se puede apreciar, los métodos de región de confianza consisten en dos partes principales. Primero, se tiene que encontrar la dirección Newton resolviendo (2.15). Después, se tiene que actualizar el radio de confianza.

Existen distintas maneras de encontrar la dirección, y siempre existe el compromiso entre precisión y eficiencia. Es decir, si se quiere calcular la dirección de una forma casi exacta es más costoso que hacer una aproximación.Una de las formas más sencillas para encontrar la dirección es encontrando el punto Cauchy que es análogo a la dirección de un gradiente descendente y consiste en encontrar el minimizador del modelo líneal

$$\mathbf{s}_l = \arg\min_{s\in\mathbb{R}^n} f(\mathbf{x}_n) + \boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n)^T\mathbf{s},$$

$$\mathbf{s}_l = -\frac{\Delta}{\|\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n)\|}\boldsymbol{\nabla}\mathbf{f}(\mathbf{x}_n).$$

Después se calcula un escalar τ que minimice al modelo cuadrático

$$\tau = \arg\min_{\tau>0} M(\mathbf{x}_n + \tau\mathbf{s}_l),$$

y el punto Cauchy estará dado por

$$\mathbf{s}_c = \tau\mathbf{s}_l.$$

Para reducir o ampliar el radio de confianza se estudia la relación (2.16).

$$\rho = \frac{f(\mathbf{x}_n) - f(\mathbf{x}_n + \mathbf{s})}{M(\mathbf{x}_n) - M(\mathbf{x}_n + \mathbf{s})} \tag{2.16}$$

Note que si ρ es negativo entonces $f(\mathbf{x}_n + \mathbf{s}) > f(\mathbf{x}_n)$ y $\mathbf{s}$ se rechaza. Si ρ se acerca a 1, entonces el modelo aproxima bien a la función y el radio de la región tiene que crecer. Si ρ es positivo pero no se acerca a 1 entonces la región no se altera.

Finalmente, si ρ es negativo o se acerca a cero, el radio de confianza debe ser más pequeño. El algoritmo de Nocedal [WhN99] para cambiar la región de confianza es el siguiente.

Algoritmo 2 *Algoritmo región de confianza*

Dado el radio inicial Δ_0
Dado el punto de inicio $\mathbf{x}_0$
Dado el número de iteraciones máximo m
for $k = 0$ to m **do**
  Obtener dirección s
  Obtener concordancia entre el modelo y la función de evaluación ρ
  if $\rho < c_1$ **then**
    $\Delta_{k+1} = c_2 \|\mathbf{s}\|$
  else
    if $\rho > c_3$ y $\|s\| = \Delta_k$ **then**
      $\Delta_{k+1} = \text{mín}(2\Delta_k, c_4)$
    else
      $\Delta_{k+1} = \Delta_k$
    end if
    if $\rho > c_5$ **then**
      $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}$
    else
      $\mathbf{x}_{k+1} = \mathbf{x}_k$
    end if
  end if
end for

Otros criterios para saber si se debe contraer o agrandar la región de confianza consisten en usar las condiciones de decremento usadas en las líneas de búsqueda.

2.1.9. Quasi-Newtons

Un aspecto costoso en el método Newton es aproximar la matriz jacobiana, si se trata de sistemas no lineales, o la matriz hessiana en el caso de optimización sin restricciones. Para lograr un menor costo computacional, se pueden utilizar métodos quasi-Newton los cuales aprovechan la información disponible hasta la

iteración actual para aproximar la matriz jacobiana o la matriz hessiana en la iteración siguiente. En esta sección se ejemplifica el comportamiento básico de esta clase de métodos. Para esto se describirá el método de Broyden para aproximar una matriz jacobiana [DS96].

La mayoría de quasi-Newton se basan en la idea que sigue el método de la secante para funciones de una variable. Éste busca aproximar la derivada de la función de evaluación por medio de dos puntos (ver figura 2.3) usando la expresión (2.17).

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \tag{2.17}$$

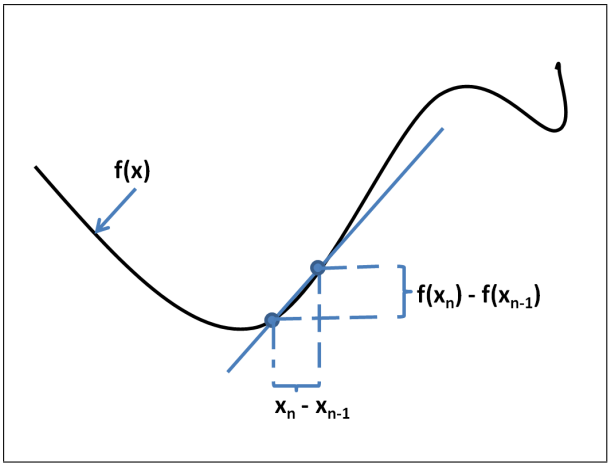


Figura 2.3: Método de la secante.

Al reformular (2.17) para encontrar la matriz jacobiana se obtiene la ecuación de la secante

$$A\mathbf{s} = \mathbf{y},$$

donde $\mathbf{s} = \mathbf{x}_n - \mathbf{x}_{n-1}$ y $\mathbf{y} = \mathbf{F}(\mathbf{x}_n) - \mathbf{F}(\mathbf{x}_{n-1})$. Ahora el problema consiste en encontrar una $A \in \mathbb{R}^{nxn}$ para la cual se satisfaga la ecuación de la secante. El método de Broyden hace la suposición de que la matriz jacobiana cambia muy poco con respecto a cada iteración y busca una A_n que minimice el cambio en el modelo lineal en la iteración n y $n-1$, es decir minimizar la expresión (2.18) sujeta a la ecuación de la secante.

$$\mathbf{M}_n(x) - \mathbf{M}_{n-1}(x) = \mathbf{F}(\mathbf{x}_n) + A_n(\mathbf{x} - \mathbf{x}_n) - \mathbf{F}(\mathbf{x}_{n-1}) - A_{n-1}(\mathbf{x} - \mathbf{x}_{n-1}) \tag{2.18}$$

La ecuación para actualizar la matriz jacobiana es la siguiente

$$A_n = A_{n-1} + \frac{(y - A_{n-1}s)s^T}{s^T s}.$$

Note que un procedimiento parecido puede hacerse para aproximar la derivada inversa de la función dada y así evitar resolver el sistema lineal para encontrar la dirección Newton (2.10,2.11). Es decir, reformular (2.17) de la siguiente manera

$$\frac{1}{f'(x_n)} \approx \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

2.1.10. Métodos híbridos

Como ya se mencionó, el método Newton suele comportarse de forma irregular cuando el punto de inicio está alejado de la solución. Las regiones de confianza resuelven este problema definiendo un radio que acota la dirección Newton. Por otro lado, las líneas de búsqueda encuentran un escalar que garantice el decremento de la función en la siguiente iteración. Otros métodos combinan el método de gradiente descendente y el método Newton para resolver el problema de un mal punto inicial.

El método de gradiente descendente consiste en tomar el punto de la siguiente iteración en dirección contraria del gradiente de la función dada.

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \lambda \mathbf{s}, \text{ con } \mathbf{s} = -\boldsymbol{\nabla}\mathbf{f},$$

donde λ es un escalar que propicia $f(\mathbf{x}_n + \lambda \mathbf{s}) < f(\mathbf{x}_n)$. La estrategia que siguen los métodos híbridos es usar el método del gradiente descendente cuando la dirección Newton es deficiente, que bien puede pasar cuando el punto inicial está lejos de la solución o cuando la matriz Jacobiana o Hessiana se aproxima a la singularidad. En esta sección se mostrará el comportamiento que siguen los métodos híbridos describiendo el funcionamiento de los métodos de *"pata de perro"*.

Los métodos de pata de perro consisten en definir una trayectoria que vaya del punto Cauchy (note que se trata del gradiente descendente dentro de la región de confianza) hacia el método Newton, y además, se acota con una región de confianza. Si el radio de confianza es pequeño, entonces el minimizador del modelo lineal, es decir el punto Cauchy, es suficiente. Si el radio de confianza es grande, se necesita el minimizador del modelo cuadrático, es decir el método Newton (ver

figura 2.4).

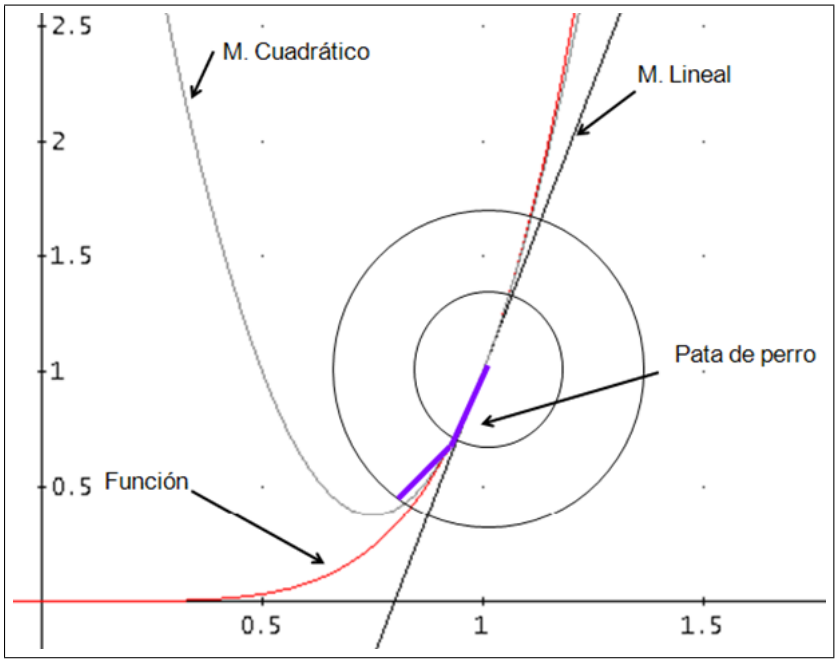


Figura 2.4: Funcionamiento de los métodos de "pata de perro".

La trayectoria entre el punto Cauchy $\mathbf{p}_c$ y la dirección Newton $\mathbf{p}_n$ suele verse como una función $p(\tau)$ cuya magnitud se incrementa conforme $M(x_n + p(\tau)\ )$ decrece. La trayectoria que define Nocedal [WhN99] es la siguiente:

$$\mathbf{p}(\tau) = \begin{cases} \tau \mathbf{p}_c & 0 \leq \tau \leq 1 \\ \mathbf{p}_c + (\tau - 1)(\mathbf{p}_n - \mathbf{p}_c) & 1 \leq \tau \leq 2 \end{cases} \tag{2.19}$$

el valor de τ se puede calcular de forma analítica resolviendo

$$\|\mathbf{p}_c + (\tau - 1)(\mathbf{p}_n - \mathbf{p}_c)\|^2 = \Delta^2.$$

Hay que notar que si $\tau = 1$, entonces la trayectoria se trata de un punto Cauchy. Cuando $\tau = 2$, la trayectoria se convierte en un Newton.

2.2. Enfoque patrón orientado a objetos

2.2.1. ¿Qué es un patrón?

> *"... en la esencia de todas las formas de creación exitosas y en la esencia de todos los procesos exitosos de crecimiento, a pesar de que hay un millón de diferentes versiones de estos procesos y formas, existe una característica fundamental invariante, que es responsable del éxito. A pesar de que ha tomado diferentes formas, en diferentes épocas y lugares, permanece inevitable e invariante a todo"* Christopher Alexander [Ale79].

El concepto de patrón surge de la arquitectura con Christopher Alexander como precursor, sus ideas acerca del diseño expuestas en el libro *The timeless way of building* [Ale79] han influido de manera significativa en el diseño de software orientado a objetos. Christopher argumenta que todas las construcciones vivas (como suele llamar a las construcciones que podrían considerarse "bellas") comparten características intrínsecas o patrones que se pueden utilizar para realizar nuevos diseños. Para Christopher Alexander: *"cada patrón describe un problema que ocurre una y otra vez en un entorno, y luego describe la esencia de la solución para ese problema, de tal forma que se pueda usar un millón de veces, sin hacer lo mismo dos veces"*. De acuerdo a lo anterior, un patrón se puede ver como una dualidad constituida por problema y solución: "*un patrón es, de forma breve, algo que pasa en el mundo y al mismo tiempo la regla que rige la creación de ese algo"*. Durante el proceso de creación, el patrón es entonces, lo que se crea y el proceso para hacerlo: la descripción de ese algo vivo y la descripción que generará ese algo.

Las ideas de Christopher fueron bien recibidas en el desarrollo de software propiciando la búsqueda de patrones en los buenos diseños de software. El primer libro acerca de patrones de software es *Design patterns: elements of reusable object-oriented software* [GHJV95] o mejor conocido como *el de la banda de los cuatro*. Para ellos, un patrón debe tener al menos las siguientes características:

- Nombre: describe el problema y su solución en una palabra o dos. Es una descripción en alto nivel que pueden entender las diferentes personas involucradas en el desarrollo de software.

- Problema: describe cuándo se debe aplicar el patrón, cuál es la problemática y el contexto de la misma.

- Solución: describe los elementos que conforman la solución, sus relaciones y resposabilidades. La solución no es descrita para un caso particular, sino que es una descripción general y abstracta que puede usarse en diversas situaciones.

Vlissides [Vli98] enfatiza que un patrón no sólo es una solución para un problema en un contexto, sino que toda descripción de patrón debe tener presente, además de nombre, problema y contexto, los siguientes elementos:

- Recurrencia: hace que la solución sea buena en diversas situaciones. Es decir, la solución no es propia de un caso particular.

- Enseñanza: brinda el conocimiento para tratar los aspectos que intervienen en el problema y da una solución buena para el mismo.

2.2.2. Ejemplo: patrón adaptador

Con la intención de dejar claro el concepto de patrón, a continuación se presenta al patrón Adaptador como ejemplo. El diagrama del patrón está en la figura (2.5). De ser necesario, puede encontrar un resumen de la notación UML en el apéndice C.

- **Nombre:** Adaptador

- **Intención:** adaptar la interfaz de un objeto, del cual no se tiene control, hacia una interfaz particular.

- **Problema:** un sistema tiene los datos y comportamiento apropiados, pero no tiene la interfaz correcta.

- **Solución:** el Adaptador provee un molde para la interfaz deseada.

- **Participantes y colaboradores:** el Adaptador adapta la interfaz de un adaptado para que concuerde con la que el cliente espera.

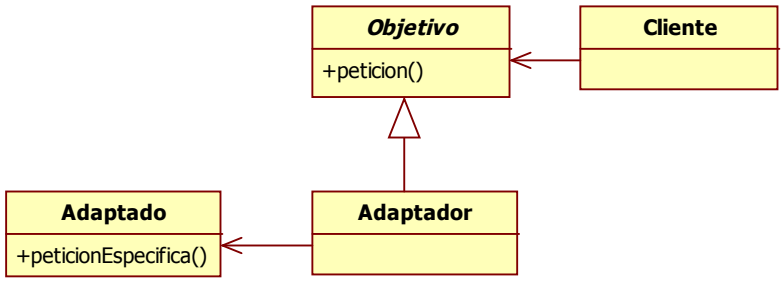


Figura 2.5: Patrón adaptador.

- **Consecuencias**: el Adaptador permite que objetos existentes puedan interactuar con nuevas estructuras de clases independientemente de sus interfaces.

- **Implementación**: introducir la clase existente dentro de otra clase y hacer que la clase contenedora concuerde con la interfaz deseada.

2.2.3. El cómo y el cuándo en los patrones

Una de las malinterpretaciones más frecuentes acerca de los patrones es que su uso lleva al éxito [Vli98]. El mal uso de los patrones puede provocar diseños difíciles de modificar e incluso una explosión de clases, es decir, la necesidad de mucho código para agregar poca funcionalidad. Por el otro lado, si cada patrón es usado de manera adecuada trae consigo dos cualidades valiosas en el software: flexibilidad y extensibilidad, es decir, la capacidad de modificar y agregar funcionalidad en el software de manera sencilla.

Para usar patrones el diseñador de software tiene que hacerse dos preguntas fundamentales: *cómo* y *cuándo* [Fow03]. El cómo suele estar dentro de la especificación del patrón, incluso se acostumbra poner ejemplos acerca de cómo usar el patrón. El problema en el cómo comienza cuando el diseño está compuesto por distintos patrones que interactúan entre sí [ST01]. Además de eso, otra piedra en el zapato es el *cuándo*. Identificar el patrón adecuado para un problema es una tarea difícil porque no existe una manera sistemática para ello, depende de un análisis profundo del problema y la perspectiva que tenga el diseñador al atacar

el problema; suele verse como un proceso creativo en donde el diseñador muchas veces se deja llevar por su intuición y experiencia.

2.2.4. Tipos de patrones

Existen distintos patrones que pueden ser clasificados por su nivel de abstracción. Por ejemplo, los patrones de análisis de Fowler [Fow97] se pueden ver como patrones de alto nivel ya que son patrones conceptuales que no están ligados a un lenguaje o enfoque de programación. Por otro lado, los patrones idioma descritos en el libro de Buschmann et al. [BMR+08], son patrones de nivel bajo ya que describen formas de usar un lenguaje de programación. Los siguientes son algunos de los tipos de patrones que se utilizan para desarrollar software, ordenados de acuerdo a su nivel de abstracción:

1. Patrones de análisis: "son grupos de conceptos que representan construcciones comunes en el modelado de negocios" [Fow97].

2. Patrones de arquitectura: se pueden ver como esquemas que expresan la estructura y organización de los sistemas de software; son diseños de alto nivel en donde se representa la comunicación entre los subsistemas que conforman a un sistema. Los patrones de arquitectura sirven como una base que se puede detallar para diseñar sistemas de diversa índole [BMR+08].

3. Patrones de diseño: son buenas soluciones, para problemas recurrentes, que se han venido presentando en el diseño de software [GHJV95].

4. Idiomas: son patrones de bajo nivel que describen la forma en que se debe hacer una implementación en un lenguaje de programación específico [BMR+08].

2.2.5. Cómo los patrones resuelven problemas de diseño

Los patrones se presentan en las buenas construcciones de software porque en ellos se plasman los principios de diseño usados por los expertos [GHJV95]. De esta manera, los patrones no sólo sirven para edificar software reusable, sino también como una forma de enseñar los principios que usan los diestros en el diseño de software [Fow03]. Se suele decir, que una vez que se aprende a usar patrones "los patrones ya no son importantes" [ST01]. Esto porque se aprende algo

más importante: los principios de diseño. Es decir, la descripción de las fuerzas (cada aspecto del problema que tiene que ser tomado en cuenta para resolverlo) y cómo tratarlas para crear buenos diseños. Los principios de diseño que son usados constantemente por los patrones se describen a continuación [GHJV95].

Delegar responsabilidades

Este principio se basa en ceder parte de las responsabilidades de un objeto a otro objeto. Lo que se quiere es que cada objeto sólo se encargue de sus responsabilidades debidamente definidas. Generalmente, delegar es un buen principio de diseño, sin embargo, su abuso puede provocar un diseño difícil de entender. La delegación funciona mejor cuando se usa basándose en estándares de diseño, es decir, en patrones.

Diseñar a interfaces y no a implementación

Una de las características fundamentales para que el enfoque OO brinde software flexible, es el polimorfismo. Diseñar a interfaces se refiere a manipular los objetos en términos de interfaces definidas por sus clases abstractas. Hacer esto permite que, por medio del polimorfismo, se puedan variar las distintas implementaciones, de una clase abstracta, en tiempo de ejecución. También permite integrar nuevas implementaciones agregando código y sin necesidad de modificar el existente.

Usar la herencia para controlar variaciones y no para hacer clases más específicas

Reutilizar código usando herencia es fácil y barato. Sin embargo, abusar de esta característica suele llevar a jerarquías de clases grandes en donde un cambio en la clase padre deriva en cambios en sus clases hijas. Esto quiere decir que existe una dependencia fuerte entre las clases padre y sus hijas, lo que limita la flexibilidad y por lo tanto el reuso. La manera apropiada de usar la herencia es hacer que todas las subclases puedan responder a una petición en la interfaz de la clase abstracta.

Favorecer a la composición antes que a la herencia

Al tipo de reuso por medio de subclases frecuentemente se le llama de "caja blanca". Esta forma de reuso generalmente rompe con el encapsulamiento, ya que la mayor parte del funcionamiento de la clase padre es visible para su hija. Otra forma de reuso se refiere a agregar funcionalidad a una clase mediante la composición de otros objetos, lo que se conoce como reuso de "caja negra".

El reuso de caja negra por medio de composición es preferible ya que dicha composición puede ser definida en tiempo de ejecución, en contraste con la forma reuso por herencia que es definido en tiempo de compilación.

Los patrones utilizan de forma adecuada los principios anteriores, lo que provoca diseños especialmente hechos para cambiar. Por lo tanto los sistemas que utilizan de forma correcta los patrones, podrán propiciar los cambios adecuados para cubrir los nuevos requerimientos durante su tiempo de vida.

2.2.6. Métricas de inestabilidad y abstracción

La flexibilidad se refiere a la capacidad del diseño para soportar los cambios, mientras que extensibilidad es la capacidad para incorporar nuevas funcionalidades. Robert Martin [Mar03] ha creado una serie de métricas para medir la flexibilidad y la extensibilidad en paquetes de software. Sus métricas se basan en el grado de inestabilidad y abstracción de cada paquete. A continuación se detalla un poco acerca de estos aspectos.

Inestabilidad

Un paquete es inestable si existe un riesgo grande de ser modificado cuando otros paquetes que conforman el sistema sean modificados [Mar03]. El paquete es más inestable mientras más interactúe con otros paquetes. La interacción entre paquetes es deseable, pero no demasiada.

Teniendo en consideración lo anterior, podemos decir que un paquete es inestable si hace uso de muchas clases fuera de ese paquete. Por el otro lado, un paquete es estable si no depende de ninguna otra clase fuera del paquete, pero hay muchas clases que dependen de éste. Nótese que un paquete inestable es propenso a sufrir modificaciones cuando las clases de las que depende sean cambiadas. En el caso de los paquetes estables, no es conveniente modificarlos porque puede

provocar muchos cambios en las clases dependientes. Para medir la inestabilidad del paquete se usa la fórmula (2.20) [Mar03].

$$I = \frac{C_e}{C_a + C_e} \tag{2.20}$$

Donde:

- C_a es el acoplamiento aferente: número de clases fuera del paquete que dependen de las clases dentro del paquete.
- C_e es el acoplamiento eferente: número de clases adentro del paquete que dependen de clases afuera de este paquete.

El paquete será estable cuando I=0, y por el otro lado I = 1 indica que el paquete es inestable.

Abstracción

La abstracción en un paquete es deseable porque permite extender o incorporar clases y por lo tanto agregar nuevas funcionalidades en el paquete. Sin embargo, las clases abstractas no pueden instanciarse, es decir no pueden utilizarse directamente, sólo sirven como una base para las clases concretas. Un paquete completamente abstracto proporcionará mucha extensibilidad pero poco funcionamiento. En el otro extremo, un paquete que no sea abstracto no permite incorporar nuevas clases concretas (funcionalidad) en el paquete. Un paquete completamente abstracto es de utilidad si sirve como la base en la que se sustentan los otros paquetes, de otra manera, ese paquete será de poca ayuda para el sistema ya que no brinda funcionalidad. Se puede decir que la abstracción es deseable en un paquete, pero no demasiada. Para medir la abstracción en un paquete se usa la fórmula (2.21) [Mar03].

$$A = \frac{N_a}{N_c} \tag{2.21}$$

Donde:

- N_a es el número de clases abstractas.
- N_c es el número de clases en el paquete.

Principio de estabilidad y abstracción

Cuando una clase es extendida se crea una dependencia entre la clase padre y la clase hija. Si la clase padre es cambiada, sus clases concretas se verán afectadas por ese cambio. Lo que se quiere es que las clases concretas estén sustentadas por una base sólida difícil de cambiar. Es decir, se pretende que los paquetes más abstractos sean los más estables, para que así su estabilidad sea una garantía para poder extender sus clases abstractas. El principio de estabilidad y abstracción establece que *un paquete debe ser tan abstracto como es estable* [Mar03]. Si se grafica tanto la inestabilidad como la abstracción, existen dos zonas ideales en donde deben situarse los paquetes (ver figura 2.6). En la zona (0,1) se deben de encontrar todos los paquetes más abstractos, mientras que en la zona (1,0) deben situarse los paquetes más inestables.

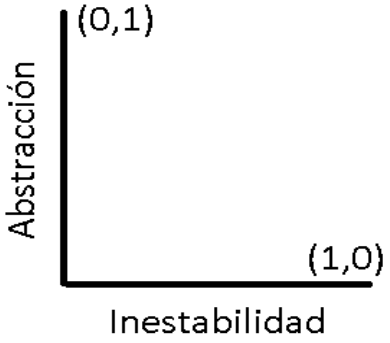


Figura 2.6: Zonas ideales en donde situar los paquetes [Mar03].

Ubicar los paquetes en las dos zonas ideales no siempre es posible, sobre todo si el paquete se modifica o extiende constantemente. Pero, siguiendo el principio de abstracción y estabilidad, se puede concluir que existen otras tres zonas que deben tomarse en consideración (ver figura 2.7). En la zona del dolor, es decir cerca de (0,0), están los paquetes que no se pueden extender dado que el paquete no es abstracto. Además, no se pueden cambiar porque son demasiado estables. En la zona poco útil (1,1) están los paquetes completamente abstractos, es decir que no tienen una función en concreto (son clases "vacías", no se pueden instanciar). Lo mejor es mantenerse alejados de las dos zonas mencionadas y permanecer cerca de la secuencia principal. Note que en la secuencia principal la abstracción del

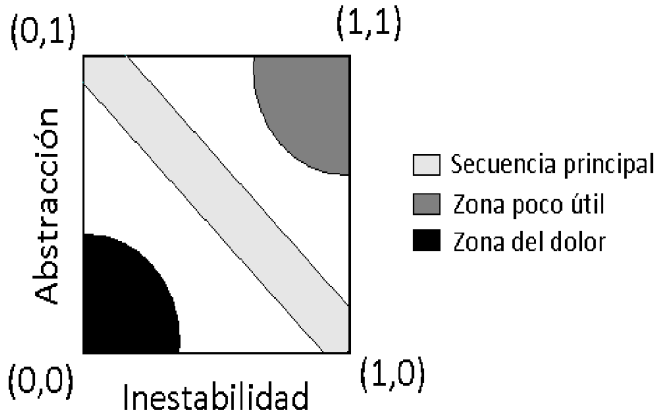


Figura 2.7: Zonas en donde pueden estar los paquetes [Mar03].

paquete está en armonía con su grado de inestabilidad.

2.2.7. Análisis de variabilidad y partes comunes

El software siempre está cambiando; ésta es una característica que lo hace inherentemente complejo. El software que no puede adaptarse a los cambios fracasa, mientras que el que puede asimilarlos tiene éxito y se hace longevo. El análisis de variabilidad y partes comunes pretende identificar las partes del software que serán propensas al cambio y las partes que permanecerán invariantes a través de la vida del software. Esto, por medio de la identificación de *familias* que comparten un concepto en común, luego viendo como varía ese concepto en cada miembro de la familia. El análisis de variabilidad y partes comunes es propuesto por Coplien [CHW02, Cop00], a continuación se da una breve explicación de este análisis.

Abstracción

La abstracción es una herramienta de análisis fundamental en el desarrollo de software. Permite ver al sistema, y sus elementos, enfocándose sólo en sus características fundamentales y dejando a un lado las propiedades poco esenciales del mismo. Es decir, permite *enfocarse en lo general y dejar a un lado lo específico.* Debido a la complejidad y el constante crecimiento de los sistemas de software, la abstracción es necesaria para poder entender al sistema durante y al término de su construcción. Para Coplien [Cop00] "*todas las técnicas de abstracción com-*

parten ciertos principios. Cada técnica muestra una manera diferente de agrupar entidades de acuerdo a las propiedades que comparten, incluyendo la forma en que cada entidad individual está cambiando". El análisis propuesto por Coplien trata de identificar familias de software que compartan propiedades comunes (análisis de partes comunes) y luego encuentra cómo difieren los miembros de cada familia (análisis de variabilidad).

Análisis de partes comunes

El análisis de partes comunes trata de encontrar aspectos en común dentro de una familia de entidades de software (en este caso objetos). Es la búsqueda de elementos para entender las relaciones entre los miembros de la familia, responde la pregunta:

> *¿Cómo es que los miembros de la familia son lo mismo?.*

Las partes en común ayudarán a crear la parte estable del diseño, es decir, el diseño que casi no se modificará y que servirá como base firme para el software. En términos de clases, los conceptos en común serán representados como clases abstractas.

Análisis de variabilidad

El análisis de variabilidad responde a la pregunta:

> *¿Cómo es que los elementos similares difieren?.*

Por lo anterior, el análisis de variabilidad sólo tiene sentido cuando hay algo común en esas variaciones. Asimismo, el análisis de partes comunes sería inútil si los elementos de una familia no difieren en algo ya que no habría razón para crear una abstracción que ayude a entender a la familia como un todo. El resultado del análisis de variabilidad es identificar las partes inestables o volátiles del software, es decir, las que serán propensas al cambio. Por lo regular, la variabilidad estará dada en clases concretas.

Matriz de análisis

Shalloway [ST01] propone una técnica sencilla para estructurar el resultado del análisis de variabilidad y partes comunes, que permite visualizar posibles patrones de software. Su técnica consiste en organizar los elementos comunes y sus variaciones en una matriz . Cada fila de la matriz será un concepto o lo común, mientras que cada columna es un escenario diferente en donde ese concepto varía (ver tabla 2.2). Por ejemplo, la variación 21 representa la variación del concepto 2 en el escenario 1.

Tabla 2.2: Forma de organizar variaciones en la matriz de análisis [ST01]

Concepto/Escenario	**Escenario 1**	**Escenario 2**	$\cdots$	Escenario N
Concepto 1	variación 11	variación 11	$\cdots$	variación 1N
Concepto 2	variación 21	variación 22	$\cdots$	variación 2N
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
Concepto N	variación N1	variación N2	$\cdots$	variación NN

Para ilustrar el uso de la matriz de análisis se usará un pequeño caso de estudio de un sistema multi-agente para agentes jugando futbol. En este sistema, los agentes tienen que coordinarse para jugar como equipo y tienen distintas formas de actuar de acuerdo al estado del ambiente. Para empezar, podemos identificar el agente en dos posibles escenarios: i) cuando el agente percibe en el ambiente que su equipo tiene la pelota, ii) cuando su equipo no la tiene. De acuerdo a esto, se pueden identificar los roles del agente con base en su percepción del ambiente, como se muestra en la tabla 2.3.

Tabla 2.3: Matriz de análisis para agentes que juegan futbol

Concepto/Escenario	**tiene el balón**	**no tiene el balón**
agente	atacar	defender

Una característica de la matriz de análisis es que puede crecer tanto como sea necesario. Por ejemplo, se puede pensar en dos tipos de agentes: los defensas y los

delanteros. También en otro nuevo escenario en donde la percepción del agente no es suficiente para determinar quién tiene el balón. La matriz quedaría como se muestra en la tabla 2.4.

Tabla 2.4: Matriz de análisis expandida para agentes que juegan futbol

Concepto/Escenario	**tiene el balón**	**no tiene el balón**	**Percepción limitada**
Defensa	atacar hasta medio campo	defender	buscar balón
Delantero	atacar	defender hasta medio campo	buscar balón

Estructurar los conceptos y sus variaciones mediante una matriz de análisis le brinda al diseñador una noción de qué patrones de software puede usar para tratar el problema. Por ejemplo, el patrón *Fábrica* está encargado de separar la creación de objetos de la lógica de programa. A su vez, está encargado de instanciar el objeto adecuado dependiendo de lo que necesite *el cliente.* Si se usa la matriz de análisis 2.4, se podría encapsular cada variación de los conceptos en una clase concreta y se concluir que el patrón Fábrica debe instanciar el objeto *atacar hasta medio campo* cuando el agente defensa sepa que su equipo tiene el balón.

Desventajas de la matriz de análisis

El verdadero problema de la matriz de análisis es poder discernir qué aspecto del problema se debe de tomar como lo común y qué es lo que difiere. Imagine un problema que involucre distintas figuras geométricas cada una con propiedades como tamaño o color. Un analista podría concluir que el concepto común está en el tamaño de las figuras, mientras que otro analista podría opinar que lo invariante es la forma de la figura y que lo que las difiere es su color y tamaño. En este aspecto, el conocimiento del problema es fundamental para decidir qué analista tiene la razón.

En particular, en el área de cómputo científico la teoría asociada a los problemas puede brindar una guía muy importante para construir la matriz de análisis. Por ejemplo, se puede pensar en el concepto Matriz y ver sus variaciones con

respecto del número de filas y columnas (ver tabla 2.5). De esta forma la teoría embona bien en la construcción de la matriz de análisis.

Tabla 2.5: Matriz de análisis para el concepto Matriz

Concepto/Escenario	**nxn**	**nx1**	**1xn**	**1x1**
Matriz	Matriz	Vector	Vector	Escalar

Capítulo 3

Estado del arte

La contribución de esta tesis toca distintas áreas. En primer lugar, se propone una arquitectura orientada a objetos y se centra en la fase de diseño por medio de patrones de software. Asimismo, la arquitectura está diseñada para lidiar con los problemas de mínimos cuadrados, optimización sin restricciones y sistemas no lineales. Debido a esto, puede ser usada en áreas como imágenes, clasificación, simulación con modelos matemáticos, etc. El aporte principal de la tesis radica en la arquitectura y el enfoque seleccionado para su construcción. Por esto, se plantearán las ventajas y desventajas del enfoque usado para construir la arquitectura comparándolo contra el enfoque tradicional (procedimental) usado para software en cómputo científico. También se describirán algunos de los trabajos más relevantes en cuanto al diseño orientado a objetos en el área de cómputo científico. Para finalizar, se enlistará la variedad de software disponible para métodos Newton y se discuten los trabajos que tienen más relación con lo propuesto en esta tesis.

3.1. Lenguajes procedimentales, enfoque OO y problema de reuso

El enfoque preferido para hacer software en cómputo científico es el procedimental, que tiene las ventajas de brindar software de alto desempeño. A continuación se comentarán a detalle algunos trabajos que describen una línea del tiempo caracterizada por problemas de reuso, asociados con lenguajes procedimentales en distintas áreas científicas. Con esto, se justifica el enfoque seleccionado

para la construcción de la arquitectura propuesta.

Dongarra y colaboradores. [DLh+94] señalan las dificultades para entender y usar el código en Fortran para matrices dispersas. Enfatizan que el formato usado para las matrices está "enredado" o acoplado con el código usado para la aplicación deseada:

> *"... código para matrices dispersas tiende a ser muy complicado debido a que el formato de los datos está atado al código de aplicación... es esencial desarrollar códigos que sean tan libres de formato como sea posible... "*

De esta manera, se pueden usar diferentes formas de representar una matriz dispersa independientemente de la aplicación deseada. Ante esta problemática, su solución fue una serie de clases en C++ para encapsular las formas de representar a las matrices y proveer interfaces más sencillas de utilizar que las subrutinas en Fortran. El diseño de sus clases sigue los siguientes principios de diseño:

- Claridad: implementación de algoritmos numéricos que se parecen a los algoritmos matemáticos en los que se basan.
- Reuso: un algoritmo en particular sólo tiene que ser codificado una vez y puede usar diferentes formas de representar una matriz.
- Portabilidad: la implementación de algoritmos numéricos puede ser usada en diferentes plataformas.
- Alto rendimiento: su librería OO tiene tan buen desempeño como C y Fortran.

Gockenbach y colaboradores [GPS99] destacan la eficiencia y robustez de los paquetes para optimización numérica como MINPACK [MGH80], MINOS [MS83] y LANCELOT [CGT92]. Sin embargo, mencionan que éstos no tuvieron el impacto que se esperaba y que inclusive muchos intentos para adaptarlos a una aplicación específica fueron un fracaso, lo que llevó a que los programadores tuvieran que escribir su propio código desde cero.

Argumentan que las causas del poco éxito de las paqueterías mencionadas fue la dificultad para adaptar la interfaz del paquete con la aplicación que necesita de

éste, la discrepancia entre las estructuras de datos utilizadas en los dos programas y que el código no representa de forma natural los algoritmos que hay detrás del mismo. Su aporte es su librería HCL (Hilbert class library) en donde definen objetos matemáticos que aparecen en el área de optimización; por ejemplo, vectores, operadores lineales, etc. Con esto, se logra código más fácil de entender y que expresa mejor los algoritmos codificados. Los autores hacen hincapié en que el enfoque orientado a objetos permite tener un grado de abstracción adecuado difícil de obtener con los lenguajes procedimentales. Asimismo, enfatizan que la vasta teoría en optimización permite identificar a los objetos de manera sencilla y que las operaciones de dichos objetos pueden estar dadas por su definición matemática.

Matthey y colaboradores [MCH+04] plantean otro problema relevante en el área de simulación de dinámica molecular: falta de software que promueva el desarrollo de algoritmos novedosos de una forma rápida y que sea apropiado para el entrenamiento de estudiantes nuevos en el área. Los autores mencionan que entornos de desarrollo como MATLAB, Mathematica y FEMLAB facilitan el desarrollo de prototipos pero no sirven para tratar con verdaderos desafíos en dinámica molecular. Una opción más tradicional para desarrollar este tipo de algoritmos es usar las librerías en Fortran como LAPACK. Sin embargo, las interfaces suelen ser muy grandes, difíciles de adaptar y si la librería soporta paralelismo (ScaLAPACK) las interfaces son aún más grandes. Su solución consiste en un framework llamado PROTOMOL que logra flexibilidad por medio del enfoque orientado a objetos y patrones de diseño. Su diseño se basa en la encapsulación de familias de algoritmos con aspectos comunes y su buen desempeño, en tiempos de ejecución, se logra por medio de *templates* en C++.

Padula y colaboradores realizan dos extensiones de HCL llamadas SVL [PSS04] y RVL [PSS09] (Standard Vector Library y Rice Vector Libary respectivamente). Su propósito principal es imitar tanto como sea posible los conceptos de cálculo en el espacio de Hilbert. El diseño de su librería está guiado por patrones de diseño de software con los cuales minimiza la dependencia en implementación y logra esconder detalles irrelevantes al programador. Esto permite codificar métodos para optimización numérica de forma limpia y natural (entre ellos métodos del tipo Newton).

Meza y colaboladores [MOHW07] retoman el problema de reuso en los paquetes para optimización pero ahora se argumenta que aunque el enfoque orientado

a objetos ha crecido (aliviando así en cierta medida el problema de reuso) aún se carece de software para propósitos generales. Plantean las necesidades, características y problemas a los que se enfrenta el usuario al usar las paqueterías de optimización:

- Los que utilizan paqueterías de optimización tienen muchos conocimientos sobre su dominio pero, generalmente, desconocen las características de los algoritmos. Los desarrolladores de los algoritmos por lo regular no se preocupan de cómo están definidos ciertos problemas, sólo se preocupan por propiedades matemáticas (Por ejemplo, continuidad en la función de evaluación)

- Un inexperto en optimización con intenciones de explorar algoritmos eficientes para resolver su problema, necesita hacer interfaces para cada paquete de optimización.

- Un desarrollador de algoritmos que desea implementar y probar su algoritmo, frecuentemente necesita hacerlo desde cero.

Su solución es una paquetería llamada OPT++ que tiene por objetivo facilitar el desarrollo, comparación y uso de métodos para optimización. Esto, definiendo interfaces comunes entre los métodos de optimización, y brindando una infraestructura para agregar nuevos algoritmos o descripciones de problemas. El diseño de su paquetería se basa en el uso del enfoque OO para crear dos jerarquías de clases: una en donde se definen los problemas no lineales y otra en donde se especifican los distintos tipos de métodos de optimización, incluyendo métodos tipo Newton y de gradiente conjugado. Hasta donde se conoce en la revisión del estado del arte en esta tesis, OPT++ es la única paquetería que trata de crear un medio común para comunicar diversos métodos Newton.

Sansalvador [San09] diseña una arquitectura para lidiar con la difícil implementación de los métodos de integración multitasa. Su arquitectura está compuesta por patrones de arquitectura y diseño, con los cuales se desacopla el modelo matemático y los métodos de integración. Además, define interfaces para incorporar diversos métodos de integración. Su arquitectura tiene la capacidad de manejar distintos pasos de integración para sistemas con diferentes escalas de tiempo.

En la tabla 3.1 se resumen los trabajos relevantes que usan el enfoque OO para cómputo científico.

Tabla 3.1: Diseño OO y patrones en cómputo científico

Autor-Año	Descripción	Área	¿Patrones?
Dongarra et al. 1994	Clases para matrices dispersas	Algebra lineal	no
Gockenbach et al. 1999	Hilbert class library (HCL)	Simulación y optimización	no
Matthey et al. 2004	PROTOMOL	Dinámica molecular	si
Padula et al. 2004	Standar Vector Library	Simulación y optimización	si
Meza et. al 2007	OPT++	Optimización	no
Sansalvador et al. 2009	Arquitectura para métodos de integración multitasa	Ecuaciones diferenciales	si
Padula et al. 2009	Rice Vector Library	Simulación y optimización	si

3.2. Acerca del compromiso entre desempeño y flexibilidad

El éxito de POO en el software de negocios se debe a que requerimientos como interacción con bases de datos, representación de documentos de negocios, operaciones con estructuras de datos complejas, etc. necesitan de software flexible y de fácil mantenimiento [CMI05]. Esto, aunado al problema de representación de fechas en el 2000 fueron un catalizador importante para que POO incursionara de forma acelerada en el software para negocios [SFG$^+$06].

A pesar de las ventajas de POO, el paradigma de programación preferido para software científico ha sido desde siempre el procedimental. Se ha evitado el uso de POO debido al costo de cómputo extra que introducen sus lenguajes [Bli02]. Sin embargo, los beneficios en cuanto a mantenimiento, flexibilidad y código de fácil entendimiento pueden ser lo suficientemente fuertes como para darles prioridad antes que un máximo desempeño en el software.

Cuando se construye software de gran envergadura la fase de mantenimiento se convierte en un aspecto vital para el éxito del software. El mantenimiento consume entre el 65 y 75 por ciento en el ciclo de vida del software [CMI05]. Pensar que

sólo el software deficiente necesita mantenimiento es una mentira: *"los productos deficientes se desechan, mientras que los buenos se reparan y se mejoran, durante 10, 15 o incluso 20 años"* [SFG+06].

El buen software OO promueve una alta cohesión[1] y un bajo acoplamiento[2] entre objetos lo que hace a cada objeto un elemento de software reusable. Cada objeto modela los aspectos relacionados con el dominio del problema, así se logra un mapeo claro entre los aspectos de la realidad y el software. Se dice que los objetos ocultan sus detalles de implementación lo que permite crear código que se puede entender mejor.

Los objetos se ven como unidades independientes que se comunican con los demás objetos por medio de su interfaz y como resultado pueden recibir mantenimiento de forma fácil y segura; reduciendo las fallas de regresión (falla en una parte del producto producida al hacer un cambio en otra parte del sistema) [SFG+06].

Aunque el software escrito en lenguajes como Fortran o C generalmente tiene mejor desempeño que el escrito en lenguajes orientado a objetos, también existe software que tiene un desempeño aceptable para sus respectivas aplicaciones. PROTOMOL [MCH+04], hace uso de programación genérica y paralelismo en C++ para desarrollar algoritmos eficientes. El desempeño de las clases de Dongarra para matrices dispersas es comparable con versiones optimizadas de Fortran 77 [DLh+94]. Clases optimizadas para manejar arreglos en Java y el uso de mejores compiladores han logrado entre un 50 y 90 por ciento del desempeño en Fortran [MMG+00]. Gockenbach [GPS99] sugiere el uso de programación mixta, dejando las partes críticas en el desempeño escritas en código procedimental, mientras que otras pueden ser escritas en C++, consiguiendo así un mejor desempeño.

3.3. Software con métodos del tipo Newton

Existen pocos trabajos en donde se trate de solucionar el problema de reuso para métodos Newton. La mayoría del software disponible con este tipo de métodos está escrito en Fortran y cuenta con las desventajas ya mencionadas. En la tabla 3.2 se resumen algunas paqueterías e implementaciones de métodos Newton. A continuación se describen los trabajos orientados a objetos, con métodos

[1] Grado de interacción dentro de un módulo [SFG+06].
[2] Grado de interacción entre dos módulos [SFG+06].

Newton, más relevantes.

Tabla 3.2: Software para métodos Newton

Nombre	Lenguaje	Tipos	Descripción
MINPACK	Fortran	Híbridos, líneas, regiones	Paquete para optimización
SMINPACK	Fortran	Híbridos, líneas, regiones	Paquete para optimización
UNCMIN	Fortran	Regiones, líneas, Quasi	Implementaciones. Opt y SENL
OPT++	C++	Regiones, líneas, Quasi	Paquete para optimización
NITSOL	Fortran	Inexactos	Implementación SENL
Sol. Non. Eq [Kel03]	MATLAB	Inexactos, Quasi	Implementación SENL
PETSC	C++	Líneas, regiones	Paquete para Ec. Diferenciales
HOMPACK	Fortran	Quasi	Homotopías
MINOS	Fortran	Quasi, líneas, regiones	Paquete para optimización
TENSOLVE	Fortran	Líneas, regiones	SENL
COOOL	C++	Lineas, Quasi-Newton	Paquete para optimización

3.3.1. PETSC

PETSC (Portable, Extensible Toolkit for Scientific Computation) contiene algunos métodos Newton para resolver sistemas no lineales, dice ser orientada a objetos pero está lejos de explotar las ventajas del enfoque OO. En lo que concierne a métodos Newton, no utiliza las características de herencia o polimorfismo para obtener código flexible.

3.3.2. COOOL

COOOL (CWP Object Oriented Optimization Library) contiene una serie de clases para operaciones con matrices y vectores, lo que brinda código limpio y entendible. En cuanto a los métodos Newton, contiene un interfaz para incorporar

distintas líneas de búsqueda. Su diseño no cubre aspectos necesarios para añadir Newtons con regiones de confianza ni quasi-Newtons.

3.3.3. OPT++

Hasta donde se ha revisado en el estado del arte, la paquetería OPT++ es la única en donde se ha tratado de crear un medio para conjuntar los diferentes métodos del tipo Newton. En su trabajo, Meza et. al. [MOHW07] definen interfaces en donde se pueden incorporar métodos quasi Newton, Newton inexactos, líneas de búsqueda y regiones de confianza. Sin embargo, su diseño carece de un análisis profundo en cuanto al método. Por ejemplo, el diseño no soporta cambios en cuanto a la condición de decremento usada en líneas de búsqueda, cambios entre los diferentes algoritmos para actualizar el radio de confianza en Newtons que usan regiones de confianza, ni cambios para utilizar diferentes técnicas de actualización de matriz en quasi-Newton. Además, su jerarquía está basada en Newtons para resolver problemas de optimización, pero no para resolver sistemas no lineales. OPT++ es un buen medio para comunicar a los métodos Newton, pero no está diseñada para lidiar con las distintas variaciones del método en diferentes problemas no lineales. Las soluciones de diseño dadas en el trabajo de tesis propuesto no pretenden competir con las de OPT++, pero tienen el potencial para hacer de OPT++ (y otras paqueterías) un software que cubra propósitos más generales.

Capítulo 4

Análisis y diseño de la arquitectura

En este capítulo se presenta el análisis de variabilidad y partes comunes realizado sobre la familia de métodos del tipo Newton, así como las soluciones de diseño derivadas de dicho análisis. Se presentan los patrones identificados, cómo éstos constituyen las soluciones de diseño y las causas que llevaron a identificar a esos patrones.

4.1. Cimientos de la arquitectura

En esta sección se describe la primera fase de la construcción de la arquitectura, que consiste en la descomposición del sistema en subsistemas y la estrategia básica para organizar la arquitectura.

4.1.1. Subsistemas y arquitectura base

La primera fase de la construcción de la arquitectura consiste en la descomposición del sistema en varios subsistemas [Som04] y está guiada por el análisis de variabilidad y partes comunes. Se quiere construir una arquitectura base que refleje la estrategia básica usada para estructurar el sistema. Pasar del análisis al bosquejo de una solución se le suele llamar *ir de la bola de lodo a la estructura* [BMR+08] y es lo que se quiere hacer en esta primera fase.

El primer concepto en común compartido por los métodos Newton está dado de forma explícita en la teoría. Éste es, la forma iterativa del método. Derivado

de la forma iterativa, se puede visualizar el siguiente algoritmo genérico [Kel03].

Algoritmo 3 *Algoritmo genérico*

while no se cumpla el criterio de paro **do**
  $s \leftarrow$ *calcular dirección Newton*
  $\lambda \leftarrow$ *calcular longitud de paso*
  $x_{n+1} = x_n + \lambda s$
end while

Los diversos tipos Newton se enfocan principalmente en las diferentes estrategias para calcular la dirección Newton o la longitud de paso. Estudiamos cómo intervienen estos dos conceptos con la finalidad de visualizar los posibles subsistemas que conformarán a la arquitectura. Una síntesis de lo anterior es presentada en la la tabla 4.1.

Apreciamos que además de la dirección Newton y la longitud de paso, otro aspecto fundamental es la función de evaluación y sus derivadas. Con esto, podemos visualizar algunos conceptos importantes en los métodos Newton a los que llamaremos componentes Newton:

- Cálculo de la dirección Newton
- Aproximación de las derivadas
- Función de evaluación
- Cálculo de la longitud de paso
- Verificación del criterio de paro

Así podemos ver que por un lado se encuentra la forma iterativa y el algoritmo genérico de los métodos, y por otro lado, están los componentes Newton. Una buena idea para la organización de la arquitectura es plasmar la forma iterativa del método ya que ésta es compartida por todos los métodos del tipo Newton. Es decir, la forma iterativa del método será la misma a lo largo de la vida del software, podemos decir que será la parte más estable del software y que, por lo

Tabla 4.1: Análisis de variabilidad y partes comunes sobre los conceptos dirección y longitud de paso

Concepto/ Escenario	Dirección	Longitud de Paso
Líneas de búsqueda	Su función principal no es encontrar una dirección Newton, sino la longitud de paso.	Es calculada para garantizar la convergencia del método. Es decir, el decremento de la función o una mejor aproximación a la solución del sistema.
Regiones de confianza	Se encuentra resolviendo el modelo lineal o cuadrático sujeto al radio de confianza.	No es dada de forma explícita; está inmersa en la dirección Newton; está acotada por el radio de confianza.
Amortiguados	Se usan métodos de eliminación Gaussiana para resolver el sistema Ax=b.	Se utiliza una línea de búsqueda para encontrarla.
Quasi-Newton	Utilizan técnicas de actualización de matriz para aproximar las derivadas. Las derivadas intervienen en la resolución del sistema Ax=b.	La longitud de paso puede ser calculada con una línea de búsqueda. También se pueden usar regiones de confianza.
Newtons inexactos o truncados	Se utilizan métodos iterativos para resolver el sistema Ax=b.	La longitud de paso generalmente es calculada con una línea de búsqueda.
Híbridos	Encuentran una trayectoria entre una dirección Newton y un gradiente descendente.	No es dada de forma explícita. Por lo regular se usa una región de confianza para garantizar convergencia.

tanto, representa una buena base de donde se pueda sustentar el software. De esta forma, un subsistema representará al algoritmo genérico y a la forma iterativa. Este subsistema será el más abstracto. Los demás subsistemas representarán los aspectos asociados con los componentes Newton. Por ejemplo, un subsistema especializado en líneas de búsqueda será lo concerniente a el componente del cálculo de la longitud de paso, otro subsistema contendrá los distintos criterios de paro, etc.

Para construir la arquitectura base se necesita estudiar la relación entre la forma iterativa y los componentes Newton. Primero, se estudia la dependencia de cada paso del algoritmo genérico con cada componente mostrado en la figura 4.1.

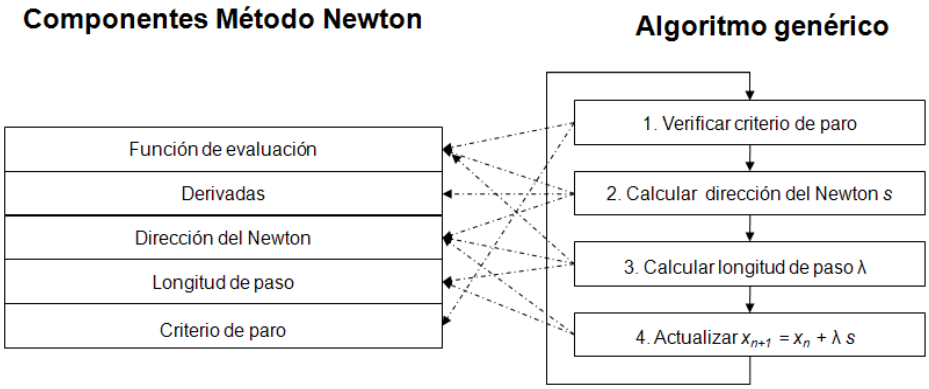


Figura 4.1: Dependencia entre cada paso del algoritmo genérico y cada componente.

Se puede ver en la figura 4.1 que existe una alta interacción entre cada paso del algoritmo y los componentes Newton. A su vez, cada componente Newton puede ser visto como un subsistema que se comunicará con el algoritmo genérico y la forma iterativa del método. El primer problema de diseño es controlar la interacción entre cada interfaz de cada subsistema y el algoritmo genérico. Para esto consideramos el patrón Fachada, descrito a continuación [GHJV95].

- **Nombre:** Fachada.
- **Intención:** provee una interfaz unificada para un conjunto de interfaces. La Fachada define una interfaz de alto nivel que hace a un subsistema más fácil de usar.
- **Aplicabilidad:**

- Cuando se quiere proveer una sola interfaz hacia un sistema complejo. Así se vuelve más sencillo de utilizar para el cliente.
- Cuando existen muchas dependencias entre el cliente y las clases de implementación de un subsistema.
- Cuando se quiere hacer una división en capas entre los subsistemas, la Fachada define una entrada hacia cada nivel del subsistema.

- **Estructura** (figura 4.2)

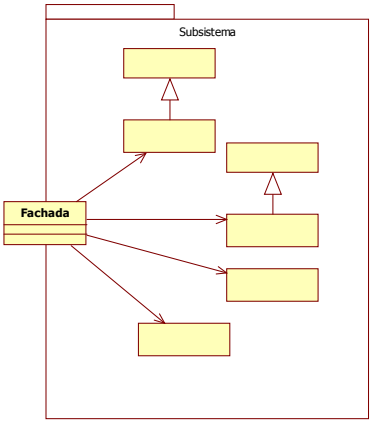


Figura 4.2: Patrón Fachada.

- **Participantes**

  - Fachada: sabe cuales clases del subsistema son responsables de la petición y delega las peticiones del cliente a los objetos correspondientes.
  - Clases del subsistema: implementan la funcionalidad del subsistema y manejan el trabajo asignado por el objeto fachada.

El patrón Fachada parece ser una solución sensata para resolver el problema de comunicación entre los componentes Newton y el algoritmo genérico. Nos gustaría que el diseño facilitara el reuso del algoritmo genérico a través de diferentes métodos. El usuario puede especificar cada componente Newton y así variar el comportamiento en cada paso del algoritmo genérico. Esto nos lleva a considerar el patrón Plantilla [GHJV95].

- **Nombre**: Plantilla.
- **Intención**: define el esqueleto de un algoritmo y permite que los pasos sean definidos mediante subclases sin cambiar la estructura del algoritmo.
- **Aplicabilidad**:
  - Se usa para implementar las partes invariantes de un algoritmo y permitir cambiar el comportamiento del algoritmo mediante subclases.
  - Cuando un comportamiento común entre subclases debe ser centralizado en una clase común para evitar código duplicado.
- **Estructura** (figura 4.3)

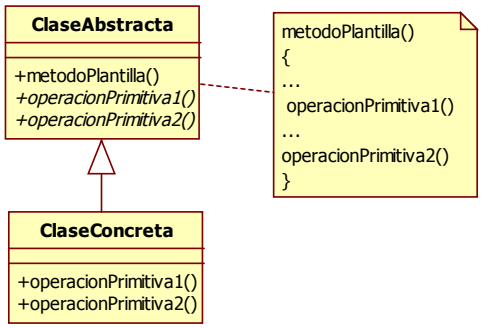


Figura 4.3: Patrón Plantilla.

- **Participantes**:
  - Clase Abstracta: define operaciones primitivas abstractas que son definidas por medio de subclases concretas e implementa un método plantilla definiendo el esqueleto de un algoritmo.
  - Clase concreta: implementa operaciones primitivas para especificar uno o más pasos del algoritmo.

Para que la arquitectura brinde la flexibilidad de poder intercambiar los distintos métodos Newton, éstos deben de compartir la misma interfaz. Asimismo,

es conveniente que el cliente no pueda ver detalles que tal vez no sean relevantes para su cometido. Sería conveniente ver a cada método como una caja negra y que una interfaz común permita variar a los distintos métodos mediante polimorfismo. Lo anterior implica separar la implementación de la abstracción en los métodos Newton, con esto se podría variar la implementación del método. Esto nos lleva a considerar el patrón Puente [GHJV95].

- **Nombre**: Puente.
- **Intención**: desacoplar una abstracción de su implementación de tal forma que las dos puedan variar independientemente.
- **Aplicabilidad:**
  - Cuando se quiere evitar un enlace permanente entre la abstracción y su implementación. Por ejemplo, cuando la implementación tiene que ser cambiada en tiempo de ejecución.
  - Tanto la abstracción como la implementación deben poderse extender mediante subclases. El patrón Puente permite variar la abstracción, la implementación y extender estas dos mediante subclases.
  - Los cambios de una implementación de una abstracción no deben tener impacto en los clientes; es decir, el código no debe ser recompilado.
  - Se quiere compartir la implementación entre distintos objetos y esto debe ser invisible al cliente.
- **Estructura** (figura 4.4)
- **Participantes**
  - Abstracción: define la interfaz de la abstracción y mantiene la referencia de un objeto de tipo Implementación.
  - AbstraccionRefinada: extiende la interfaz definida por Abstracción.
  - Implementación: define una interfaz para las clases que contienen la implementación. Las interfaces entre Abstracción e Implementación pueden diferir.

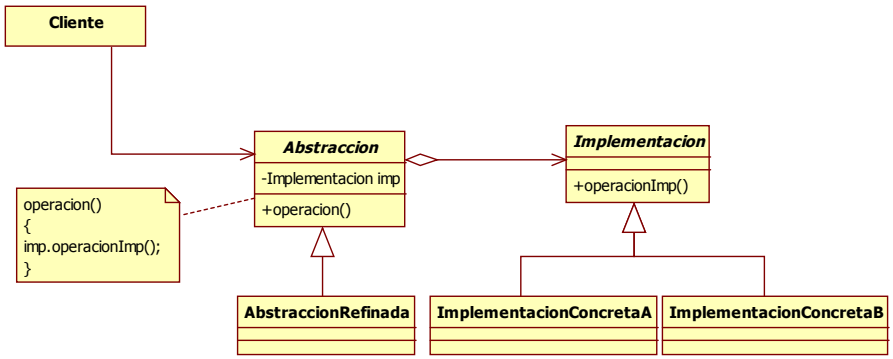


Figura 4.4: Patrón Puente.

Hasta este momento se cuenta con tres patrones que interactúan entre sí. En el patrón Puente se puede ver al método de la forma más abstracta, mientras que el patrón Plantilla permite instanciar los diversos métodos Newton variando los pasos del algoritmo genérico, el patrón Fachada permite controlar la comunicación con los componentes Newton. Podemos notar un diferente grado de abstracción entre los patrones, siendo el patrón puente la forma más abstracta del método, el patrón Fachada el que da más detalles y el patrón Plantilla situado entre los dos; ver figura 4.5 . Esta característica nos lleva a considerar un diseño en capas, en donde cada capa tiene un diferente grado de abstracción y está comunicada con las capas adyacentes.

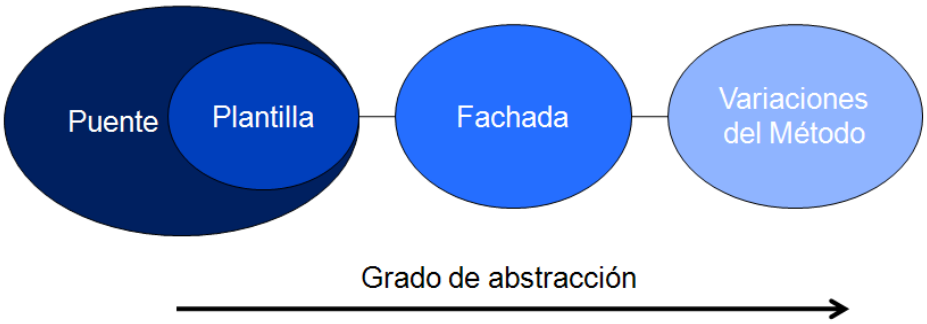


Figura 4.5: Grados de abstracción de los patrones.

4.1.2. Arquitectura en capas

El diseño en capas se refiere a la descomposición de un sistema por medio de diferentes niveles de abstracción. A cada nivel se le llama capa y depende de las capas adyacentes a ella. Buschmann *et al.* [BMR+08] ven a la descomposición en capas como un patrón de arquitectura que sirve para diseños en donde se tengan que mezclar características de bajo nivel con aspectos de alto nivel; también define una serie de pasos para realizar una descomposición en capas (no se tienen que usar todos). Los tres primeros pasos son los siguientes:

1. Definir criterio de abstracción.
2. Determinar el número de niveles de abstracción.
3. Nombrar a cada capa y especificar tareas a cada una de ellas.

A continuación se especifican estos tres aspectos en cada capa, comenzando con la capa de más alto nivel.

4.1.3. Capa 4: Newton Genérico

- El propósito de esta capa es servir como una caja negra que oculte el funcionamiento interno del método Newton.
- Esta capa se puede ver como el nivel más alto de abstracción.
- Esta capa puede proporcionar los servicios de distintos métodos Newton; brinda la flexibilidad de intercambiar diversos métodos de forma sencilla.
- La comunicación entre la capa 4 y 3 es llevada a cabo por medio del patrón Puente.
- La capa Newton Genérico será utilizada por los posibles clientes (Por ejemplo, una interfaz gráfica de usuario o un paquete de optimización numérica)

4.1.4. Capa 3: Métodos Newton

- En esta capa se especifica el algoritmo general que utilizan los métodos Newton.

- Esta capa será usada para implementar (haciendo uso de las capas 2 y 1) el método Newton que cubra mejor las necesidades del usuario.
- En esta capa se define el comportamiento de cada paso del algoritmo genérico, es decir, la forma de calcular la longitud de paso, el criterio de paro y la dirección Newton.
- La comunicación con la capa 2 se logra con el patrón fachada.

4.1.5. Capa 2: Componentes Newton

- Esta capa funciona como una interfaz hacia las distintas variaciones del método Newton.
- Se puede ver como un enlace entre la capa 3 y 1.
- El objetivo de esta capa es lograr un mejor control sobre las variaciones del método Newton definiendo una interfaz común hacia los componentes Newton.
- El patrón Fachada desacopla los componentes Newton del algoritmo genérico y brinda una interfaz unificada que es más sencilla de usar por la capa 3.

4.1.6. Capa 1: Variaciones método Newton

- Esta capa está en el nivel más bajo de abstracción. Se definen interfaces para cada una de las variaciones del método.
- Un usuario experimentado puede utilizar esta capa para hacer implementaciones particulares. Por ejemplo, puede implementar distintos algoritmos para calcular la dirección Newton o la longitud de paso, puede intercambiar la función de evaluación, etc.

El diseño final de la arquitectura base se muestra en la figura 4.6.

Como se puede notar, en la capa 1 de la arquitectura base no aparece una interfaz referente al cálculo de las derivadas. Esto es porque el diseño para el cálculo de las derivadas es un problema complejo que cambia dependiendo de la forma de la función de evaluación y el problema no lineal que se quiera tratar. Agregar dicha interfaz en la capa 1 puede complicar el diseño, se optó por delegar

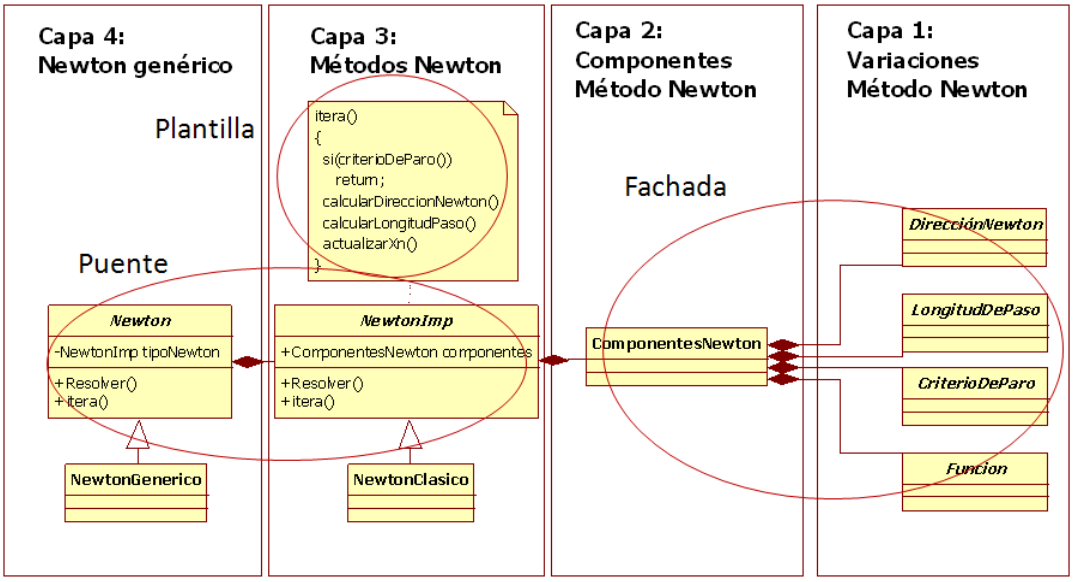


Figura 4.6: Arquitectura base.

la responsabilidad del cálculo de las derivadas a otras dos familias de clases. Una familia encargada de variar el comportamiento de la función de evaluación dependiendo del problema no lineal (descrita en la sección 4.2) y otra familia encargada de brindar una interfaz común para algoritmos de diferencias finitas, algoritmos de actualización de matrices usadas en quasi-Newton y que también puede ser usada para incorporar derivadas de forma analítica (descrita en la sección 4.3.3).

En la siguiente sección se trata el diseño asociado a los problemas no lineales y los cambios en las derivadas debido a dichos problemas.

4.2. Problemas no lineales y patrón Estado

Uno de los principales problemas en el software para cómputo científico es la falta de software para propósitos generales [MOHW07]. Es decir, software que pueda utilizarse en diversas aplicaciones o que tenga la capacidad para poderse modificar y adaptarse para distintos propósitos. En cuanto a los métodos Newton, se puede decir que la cualidad de "general" o "genérico" es dada en gran parte por la capacidad del software para adaptarse de acuerdo a los distintos problemas no lineales. Por ejemplo, métodos Newton en el área de imágenes y reconocimiento de

patrones suelen usarse para resolver problemas de mínimos cuadrados, mientras que en simulación con modelos matemáticos se usa para resolver sistemas de ecuaciones no lineales. El software que tenga la capacidad de utilizarse para distintos problemas no lineales, también tendrá las facilidades para usarse en distintas áreas de aplicación.

Habitualmente, cada método suele estar implementado específicamente para tratar el problema no lineal deseado. Esto es lamentable debido a que las similitudes entre los distintos métodos Newton ya han sido estudiadas teóricamente [DS96] y pueden ser aprovechadas para crear software que se pueda adaptar entre los distintos problemas no lineales. En esta sección se describen las ideas de diseño usadas en la arquitectura propuesta para lidiar con diferentes problemas no lineales.

Los tres problemas tratados en esta tesis tienen características íntimamente relacionadas. Hay que recordar, por ejemplo, que la resolución de un sistema no lineal puede verse como un problema de mínimos cuadrados ya que las raíces del sistema concuerdan con los mínimos correspondientes. Esto se logra al elevar al cuadrado cada función componente de la función original - transformación característica en el problema de mínimos cuadrados. Ahora bien, al realizar dicha transformación, la función de evaluación es parecida a una tratada en un problema de optimización sin restricciones.

Notamos de inmediato la relación entre los problemas no lineales, la forma de la función de evaluación y la transformación característica de mínimos cuadrados. De aquí, el inicio de nuestro análisis de variabilidad y partes comunes, en donde se estudian las variaciones entre las distintas funciones para cada problema no lineal (primer renglón de la matriz 4.2) . Al organizar los tipos de funciones para cada problema no lineal, se pueden distinguir al menos otros dos conceptos que variarán de acuerdo al escenario en el que se encuentren: la primera y segunda derivada; ver tabla 4.2.

Hasta el momento la matriz 4.2 sirve sólo como una forma para organizar los conceptos teóricos involucrados en el problema. Sin embargo, note que los aspectos comunes pertenecen a los elementos implicados en el modelo lineal y cuadrático usados en los métodos Newton. Ahora se pueden visualizar objetos como función, primera derivada y segunda derivada que esconden detalles irrelevantes al programador y que están de acuerdo con su definición matemática. Con esto, se facilita la construcción de los modelos ya mencionados.

Tabla 4.2: Matriz de análisis problemas no lineales. (SNL: sistema no lineal. OSR: optimización sin restricciones. MC: mínimos cuadrados)

Concepto/Escenario	**SNL**	**OSR**	**MC**
Función	$F : \mathbb{R}^n \rightarrow \mathbb{R}^n$	$f : \mathbb{R}^n \rightarrow \mathbb{R}$	$F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, $n < m$ $f = \frac{1}{2} \lVert F \rVert^2$
Primera derivada	$JF(x)$	$\nabla f(x)$	$\nabla f(x) = JF(x)^T F$
Segunda derivada	-	$Hf(x)$	$Hf(x) = JF(x)^T JF(x) + \sum_{i=1}^{m} f_i \, Hf_i(x)$

Por ahora, bien se podría optar por tomar cada concepto como una clase abstracta y hacer una clase concreta con cada variación de dichos conceptos. No obstante, nos interesa hacer un análisis más profundo en cuanto a los aspectos comunes en los problemas no lineales y cómo éstos son utilizados (o podrían utilizarse) en la implementación de un método Newton.

Comenzamos por describir un caso simple de un método de gradiente descendente que podría ser usado para resolver los tres problemas no lineales concernientes. Para este tipo de métodos se requiere el cálculo de la dirección $d = -\nabla f$, donde $f = \frac{1}{2} F^T F$ en sistemas no lineales y mínimos cuadrados. Suponiendo que se quiere usar el gradiente para resolver un sistema de ecuaciones, es necesaria la transformación propia de un problema de mínimos cuadrados. Por lo tanto, durante una parte del método es necesario ir de un sistema no lineal a un problema de mínimos cuadrados como se ilustra en la figura 4.7.

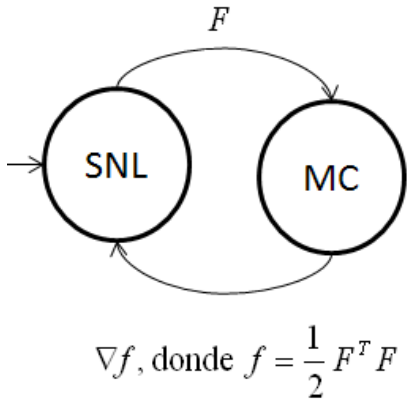


Figura 4.7: Transición de problemas no lineales en un gradiente descendente. ∇f es el gradiente de f (SNL: sistema no lineal. MC: mínimos cuadrados).

Como ya se comentó, los problemas de mínimos cuadrados y optimización sin restricciones son similares; se puede pensar en un método para optimización sin restricciones pero que pueda ser usado para mínimos cuadrados, sólo se tienen que calcular sus derivadas de manera adecuada, como se muestra en la figura 4.8.

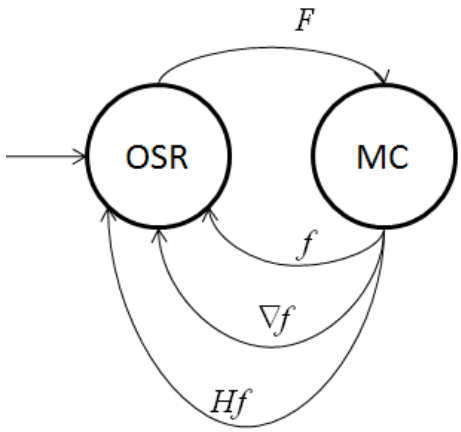


Figura 4.8: Posible transición entre optimización sin restricciones (OSR) y mínimos cuadrados (MC). Hf es la matriz hessiana de f.

Podemos ver la interacción entre los distintos problemas no lineales como una "máquina de estados finitos" en donde cada estado es un problema no lineal y la transición representa un elemento del problema. No tomamos de manera estricta la definición de máquina de estados, sólo usamos la noción de "*cambio de estado*" como una idea de diseño para lidiar con los problemas no lineales.

Para continuar con el análisis consideramos el caso del método Híbrido de Powell [Pow70], el cual es uno de los primeros métodos híbridos para tratar sistemas no lineales. En dicho método están basados otros como el de Blue [Blu80] y el que se encuentra en la paquetería MINPACK. Este método se puede considerar una "pata de perro" ya que involucra dos direcciones que están sujetas a un radio de confianza. Dependiendo de que tan grande sea el radio de confianza el método utiliza la dirección de un gradiente descendente, una dirección Newton o una "mezcla" de la dos. En ocasiones es necesario calcular el gradiente de la función usando la transformación propia de mínimos cuadrados. En otras circunstancias, se calcula una dirección Newton para sistemas no lineales, con esto, podemos visualizar una "máquina de estados" como la mostrada en la figura 4.9.

Dennis [DS96] sugiere al modelo cuadrático propio del problema de mínimos

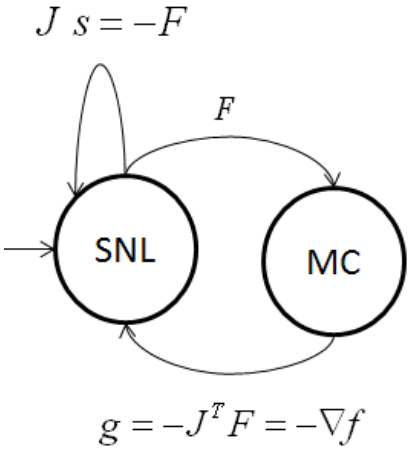


Figura 4.9: Cálculo de direcciones en el método híbrido de Powell. J es la matriz jacobiana y s es la dirección a encontrar.

cuadrados como un modelo genérico que también puede ser usado para sistemas no lineales. Esto es posible dada la similitud, ya mencionada, entre los dos problemas. Con esto, concluye que los algoritmos de líneas de búsqueda y regiones de confianza usados en optimización sin restricciones también pueden ser usados para sistemas no lineales. Sugiere el uso de $f = \frac{1}{2}F^T F$ para buscar el decremento suficiente en líneas de búsqueda y el modelo de mínimos cuadrados, sujeto al radio de confianza, en métodos de regiones de confianza. Sin embargo, la transición entre los problemas no es completa, se sugiere conservar la dirección Newton original para sistemas no lineales en lo que sea posible. Una vez más aparece esta mezcla de problemas no lineales inmersa en los métodos. Así por tanto, la transición en líneas de búsqueda cuando se quiera verificar si un punto tiene suficiente decremento será como se muestra en la figura 4.10. La transición entre problemas cuando se quiere calcular la dirección Newton en regiones de confianza puede verse como en la figura 4.11.

La idea de cambios de estado tiene potencial como un diseño flexible ya que facilita el tratar a la función de evaluación de acuerdo a un problema no lineal, así como el intercambio de problemas que, como ya se vio, se presenta en diversos métodos Newton o de optimización.

Ahora podemos ver a *problema no líneal* y a *función* como dos posibles clases altamente relacionadas. Note que la función es tratada de distintas maneras de acuerdo al problema no lineal en el que se encuentre. El *problema no lineal* es el

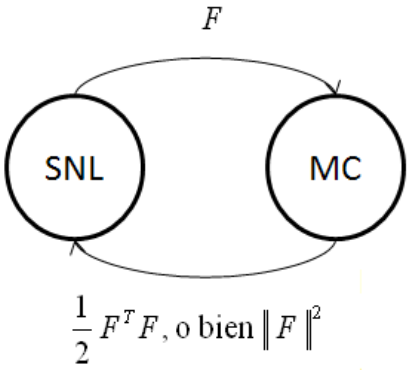


Figura 4.10: Transición entre problemas para línea de búsqueda.

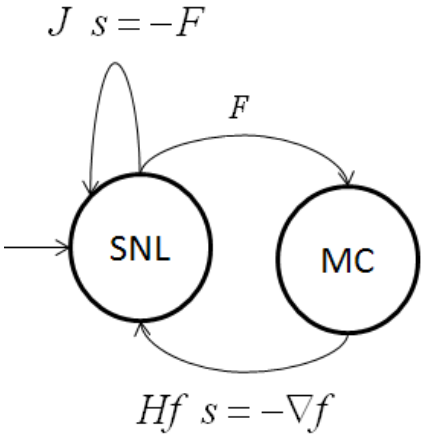


Figura 4.11: Transición de estados para calcular la dirección Newton en regiones de confianza.

estado o contexto en el que se encuentra la función de evaluación, y el *comportamiento* de la función varía de acuerdo a su contexto. Esto nos lleva a considerar el patrón Estado [GHJV95] como una solución al problema de cambios de estado y variaciones en el comportamiento interno de la función de evaluación.

- **Nombre:** Estado.
- **Intención**: permite que un objeto altere su comportamiento cuando su estado interno cambie. El objeto parecerá que cambia de clase.
- **Aplicabilidad**: el patrón Estado se puede usar en el siguiente caso.

- El comportamiento de un objeto depende de su estado y éste debe cambiar su comportamiento en tiempo de ejecución dependiendo de su estado.

- **Estructura** (figura 4.12)

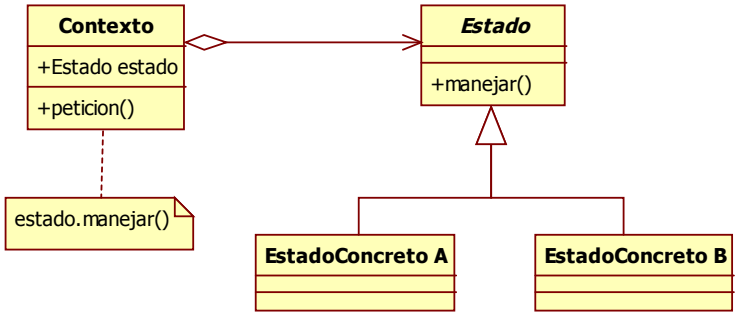


Figura 4.12: Patrón Estado.

- **Participantes**

  - Contexto
    - Define la interfaz apropiada para el cliente.
    - Tiene una instancia del estado concreto que define el estado actual.
  - Estado
    - Define una interfaz para encapsular el comportamiento asociado a un estado particular de Contexto.
  - EstadoConcreto
    - Cada subclase implementa un comportamiento asociado a un estado del contexto.

Podemos ver que cada problema no lineal es un estado concreto que define el contexto de la función de evaluación. *Funcion* será la interfaz que vea el cliente y proporcionará las derivadas de acuerdo al problema no lineal en donde se encuentre. El diseño al aplicar el patrón Estado es el de la figura 4.13. Cada estado concreto implementa las características mostradas en la matriz 4.2.

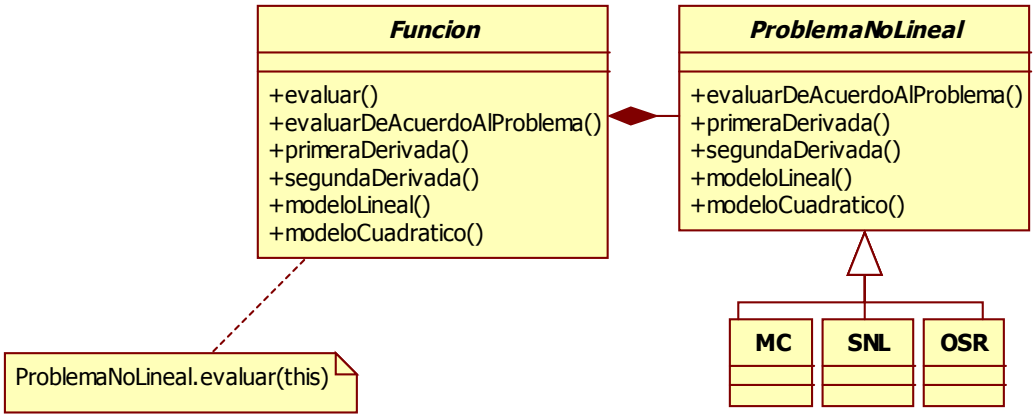


Figura 4.13: Patrón Estado para problemas no lineales.

4.3. Diseño Detallado

En esta sección se describirán las ideas de diseño que siguen los subsistemas que interactúan con la arquitectura base. El diseño se enfoca en técnicas de líneas de búsqueda, regiones de confianza, diferencias finitas para calcular derivadas y actualización de matrices usadas en quasi-Newton. Se dan detalles acerca del diseño para la creación de objetos y la interacción con paquetes externos.

4.3.1. Líneas de búsqueda

Las líneas de búsqueda son técnicas que garantizan la convergencia en los métodos Newton desde cualquier punto de inicio. Su funcionamiento básico consiste en buscar un conjunto de longitudes y luego verificar si una longitud hace que la función decrezca lo suficiente.

Para buscar una longitud de paso se suelen utilizar técnicas de bisección o interpolación; también se necesita de la evaluación de la función en diferentes puntos, así como la disponibilidad de sus derivadas. En cuanto a las condiciones de decremento, algunas de las más usadas son la condición de curvatura, Wolfe y Goldstein (descritas en el marco teórico); esta parte del algoritmo generalmente requiere de algunas operaciones aritméticas y una condición lógica.

Los dos conceptos que tomamos para el diseño son precisamente los anteriores; sus variaciones están en la tabla 4.3.

Tabla 4.3: Conceptos y sus variaciones en líneas de búsqueda

Concepto/Variaciones	Variaciones
Estrategia para buscar longitud	Bisección
	Interpolación cuadrática
	Interpolación cúbica
Condición de decremento	Wolfe
	Goldstein
	Condición de curvatura

Podemos ver a los dos conceptos como dos algoritmos que pueden varíar para crear diferentes tipos de líneas de búsqueda. Estos algoritmos tienen la particularidad de que uno está inmerso en otro. La estrategia para buscar la longitud usará la condición de decremento para saber si la longitud es adecuada o si necesita buscar otra. Lo que se quiere es poder variar la estrategia para buscar la longitud y que al mismo tiempo se pueda variar la condición de decremento.

Una buena solución para encapsular una familia de algoritmos es el patrón Estrategia. Este patrón ya se ha utilizado en el área de cómputo científico. Por ejemplo, en simulación por medio de dinámica molecular [MCH+04] y ecuaciones diferenciales [San09]. Lo que se pretende para el diseño de líneas de búsqueda es diseñar una estrategia "doble"; una inmersa dentro de la otra. En primera instancia esto daría la impresión de un diseño complicado. Sin embargo, la estrategia para la condición de decremento por lo regular será muy pequeña y fácil de manipular, lo que da la certeza para encapsularla dentro de la estrategia para buscar la longitud de paso. A continuación está la descripción del patrón [GHJV95].

- **Nombre**: Estrategia.
- **Intención**: define una familia de algoritmos, encapsula cada uno, y permite el intercambio entre los mismos. El patrón Estrategia permite intercambiar algoritmos independientemente del cliente que los use.
- **Aplicabilidad**:
  - Cuando muchas clases difieren sólo en su comportamiento.
  - Cuando se necesiten diferentes variantes de un algoritmo.

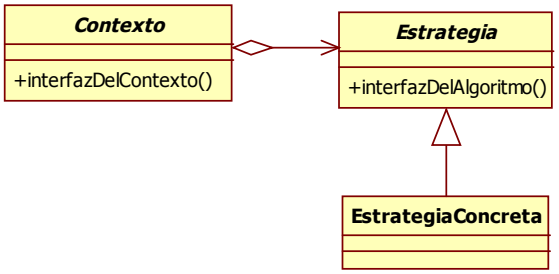


Figura 4.14: Patrón Estrategia.

- **Estructura** (figura 4.14)

- **Participantes**

  - Estrategia
    - Declara una interfaz común para la familia de algoritmos. Contexto usa esta interfaz para llamar al algoritmo definido por EstrategiaConcreta.
  - EstrategiaConcreta
    - Implementa un algoritmo usando la interfaz de Estrategia.
  - Contexto
    - Es configurado con un objeto de EstrategiaConcreta.
    - Mantiene una referencia a un objeto Estrategia.
    - Puede definir una interfaz para permitir que Estrategia accese su información.

El diseño final para líneas de búsqueda está en la figura 4.15.

4.3.2. Regiones de confianza

Las regiones de confianza garantizan la convergencia global en los métodos Newton acotando la dirección Newton a un radio de confianza. El radio de confianza se actualiza dependiendo de la concordancia que exista entre el modelo y

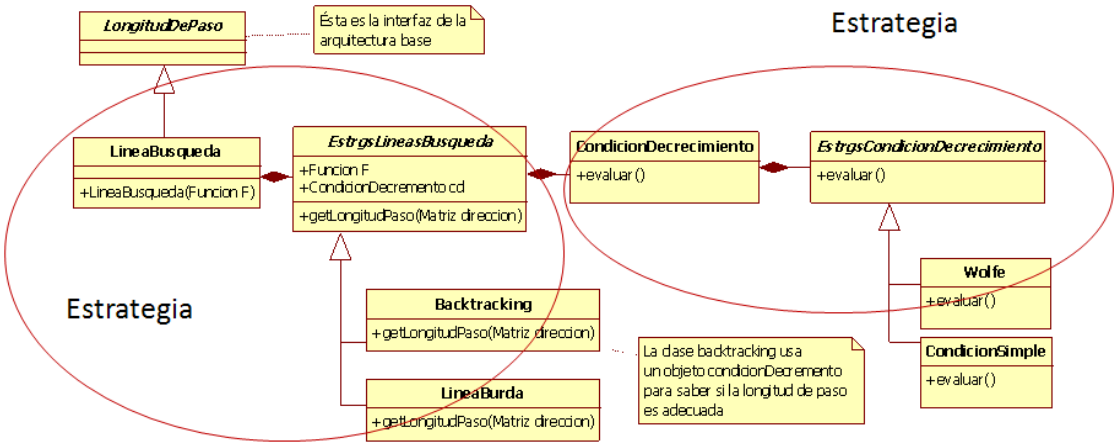


Figura 4.15: Diseño de líneas de búsqueda.

la función tratada. Para encontrar la dirección Newton es necesario resolver un problema de minimización con restricciones, lo cual puede hacerse de diferentes formas. Tomando en consideración lo anterior, los conceptos a tener en cuenta en el diseño están en la tabla 4.4.

Tabla 4.4: Conceptos y variaciones en regiones de confianza

Concepto/Variaciones	**Variaciones**
Solución del problema con restricciones	Punto Cauchy.
	Métodos pata de perro o *dogleg*.
	Mínimización de subespacio de dos dimensiones.
Actualización de radio de confianza	Métodos adaptativos usando umbrales.
	Método de Dennis
Concordancia entre modelo y función	Razón entre la función y el modelo.
	Condición de suficiente decremento.

Queremos controlar las variaciones de los tres conceptos dados en la tabla 4.4. Se tiene que proceder con cautela al manejar tantas variaciones que interactúan entre sí. Optar por una solución como la manejada en el diseño para líneas de búsqueda puede no ser tan conveniente. El patrón estrategia utiliza la delegación de responsabilidades para variar el comportamiento de una clase. El uso indiscriminado de este principio puede provocar diseños difíciles de entender y problemas para la comunicación entre objetos.

Por lo anterior, se optó por buscar un diseño lo más sencillo posible, que use la delegación de responsabilidades de forma mesurada. Los algoritmos para la solución del problema con restricciones se pueden ver como una familia de clases que comparten una misma interfaz. Entonces, se puede pensar en una clase abstracta para el concepto y en clases concretas que definan la implementación de cada variación del algoritmo. La concordancia puede ser encapsulada en un método abstracto, de tal forma que el usuario pueda sobrescribirlo para usar el criterio de concordancia que guste; ver figura 4.16.

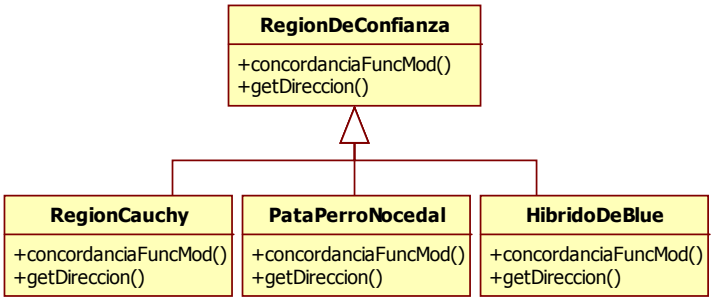


Figura 4.16: Diseño para la solución del problema con restricciones y concordancia entre modelo y función.

Para variar la manera de actualizar el radio de confianza se puede utilizar el patrón Estrategia. Esto es sensato ya que la familia de clases de la figura 4.16 tiene la carga de dos variaciones, agregar otra variación puede provocar clases demasiado grandes. En esta situación delegar parece razonable y se puede hacer mediante el patrón Estrategia.

Otro aspecto que se presenta de forma frecuente en métodos de regiones de confianza, es el uso de direcciones que son "ajustadas" al radio de confianza. Por ejemplo, el punto Cauchy y el minimizador a lo largo de un modelo líneal son direcciones de un gradiente descendente acotadas por un escalar (puede ser el radio de confianza). Por medio del diseño presentado en la sección 4.2 para lidiar con los problemas no lineales, se pueden implementar diferentes direcciones de forma sencilla. A su vez estas direcciones pueden ser encapsuladas en clases. Nos gustaría reutilizar estas direcciones para implementar algoritmos de regiones de confianza. Esto involucra, por un lado, ajustarlas al radio de confianza, y por otro, adaptarlas a una interfaz especial para direcciones de regiones de confianza.

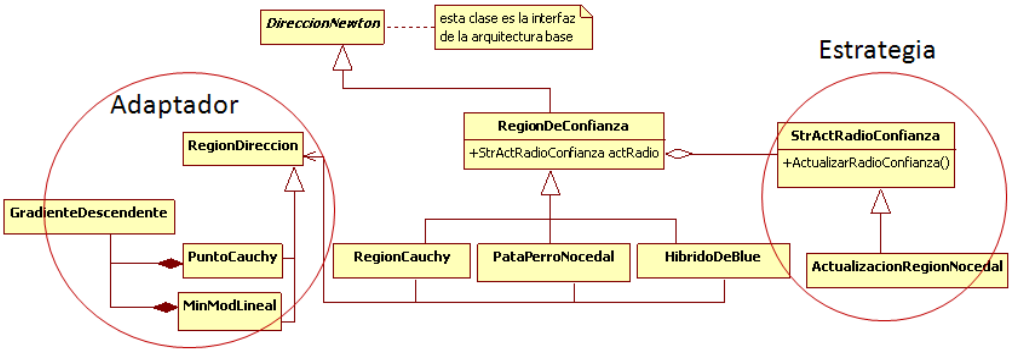


Figura 4.17: Diseño para regiones de confianza.

Consideraremos el patrón Adaptador cuya descripción fue dada en el marco teórico (subsección 2.2.2).

El diseño para las regiones de confianza se muestra en la figura 4.17. El patrón Adaptador adapta una interfaz especial para que los algoritmos que resuelven el problema sin restricciones (*RegionCauchy, PataPerroNocedal*, etc.) puedan reutilizar direcciones ya existentes. El patrón Estrategia en esta ocasión permite variar los algoritmos de actualización de radio de confianza.

4.3.3. Derivadas

El diseño debe proveer un medio para incorporar las diferentes maneras para aproximar derivadas, así como un medio para incorporar las derivadas analíticas. Las derivadas pueden tener diferentes formas, por ejemplo, un gradiente, una matriz jacobiana o una matriz hessiana, y pueden ser calculadas de diversas formas; ver figura 4.18.

El criterio para construir la familia de clases y la interfaz en donde se puedan incorporar las derivadas es dado por medio del análisis de variabilidad de partes comunes en la tabla 4.2. Habrá dos familias de clases: una que defina una jerarquía para derivadas de primer orden y otra para derivadas de segundo orden. Estas familias serán de ayuda para que el patrón Estado, encargado de lidiar con los problemas no lineales, pueda calcular las derivadas apropiadas de acuerdo a la forma de la función de evaluación. Por lo tanto, las clases *DerivadaPrimerOrden* y *DerivadaSegundoOrden* serán agregados de ProblemaNoLineal; ver figura 4.19.

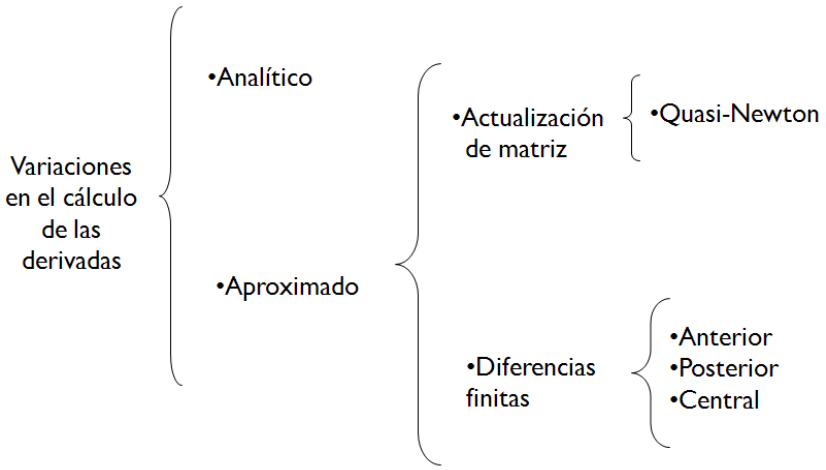


Figura 4.18: Variaciones en el cálculo de las derivadas.

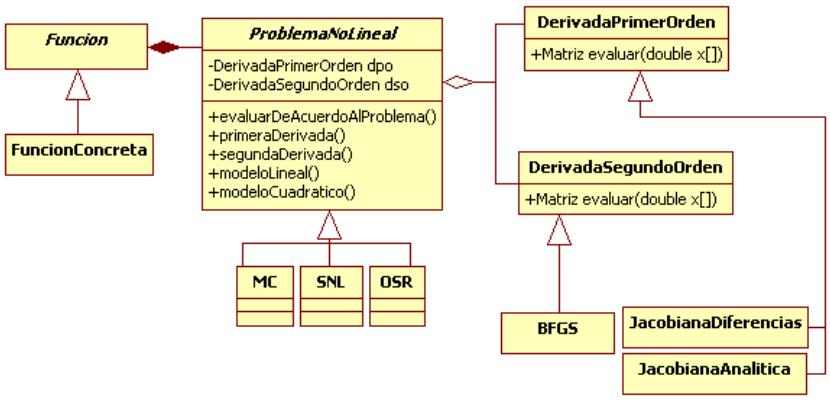


Figura 4.19: Interacción entre las derivadas y el patrón Estado.

4.3.4. Creación de objetos

Es importante separar la creación de objetos de la lógica del programa. Así se puede variar la creación de objetos independientemente de los algoritmos. Considere, por ejemplo, el caso de un usuario que desea medir el impacto que tienen, en un algoritmo, el cálculo de derivadas por medio de diferentes criterios. El usuario ha creado una familia de clases con los distintos criterios de derivadas, todos ellos comparten la misma interfaz de tal manera que se puedan intercambiar mediante polimorfismo; sin embargo, dentro del algoritmo indica cual es la implementación específica para crear las derivadas. En el mejor de los casos, el usuario habrá hecho lo anterior una vez y modificará y recompilará tantas veces como número de criterios con los que quiera hacer pruebas. En el peor caso, el usuario habrá especificado, demasiadas veces, supongamos 15, cual es el objeto que debe de crearse para calcular las derivadas. Si desea hacer pruebas con 3 criterios, tendrá que modificar 45 partes del código.

Lo anterior puede ser desastroso si en vez de un algoritmo se trata de un sistema con miles de líneas de código. Lo mejor es tener un sólo lugar para indicar cuál es el objeto que se tiene que instanciar por cada familia de objetos. A este lugar recurrirán los algoritmos cuando necesiten de un objeto. De esta forma, para variar la creación de objetos se modificará sólo un fragmento de código y los algoritmos codificados permanecerán intactos. Esta idea es retomada en la arquitectura por medio del patrón Fábrica Abstracta [GHJV95].

- **Nombre:** Fábrica Abstracta.
- **Intencion:** provee una interfaz para crear familias de objetos relacionados sin especificar su clase concreta.
- **Aplicabilidad:**
  - Cuando un sistema debe ser independiente de cómo sus productos son creados, compuestos o representados.
  - Cuando un sistema debe ser configurado con un producto dentro de una familia de productos.
  - Cuando se quiere proveer una librería de clases y sólo se quiere dar a conocer su interfaz pero no su implementación.

- **Estructura** (figura 4.20).

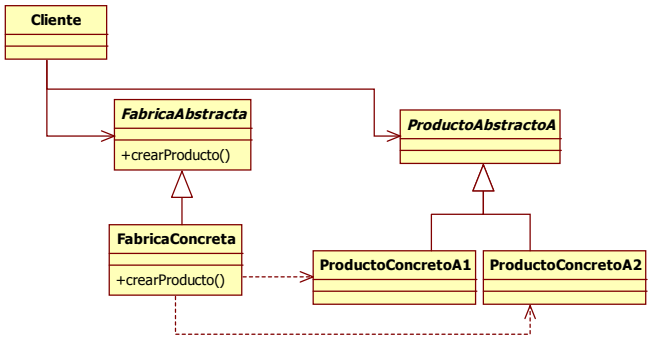


Figura 4.20: Patrón Fábrica Abstracta.

- **Participantes**

  - FabricaAbstracta: declara una interfaz para operaciones que crean productos objeto.
  - FabricaConcreta: implementa las operaciones para crear un producto objeto concreto.
  - ProductoAbstracto: declara una interfaz para un tipo de producto objeto.
  - ProductoConcreto: define un producto objeto que será creado por la fábrica concreta correspondiente.
  - Cliente: usa una sola interfaz declarada por FabricaAbstracta y ProductoAbstracto

En nuestro caso, la Fábrica Abstracta no sólo puede servir para separar la creación de objetos de los algoritmos, sino también es la forma de instanciar diferentes métodos del tipo newton. Es decir, mediante la Fábrica se le puede indicar a la arquitectura los diferentes aspectos que intervienen en ella. Por ejemplo, el tipo de dirección que debe usar, la estrategia de línea de búsqueda, la forma de calcular las derivadas. Con esto, la arquitectura base se convierte en una estructura que puede tomar diferentes formas o instancias por medio de la Fábrica Abstracta.

La creación de objetos se necesita prácticamente en toda la arquitectura. Nos gustaría que el diseño permitiera el fácil acceso a las fábricas lo cual puede ser logrado por medio del patrón *Singleton*.

- **Nombre:** *Singleton.*
- **Intención:** se asegura que una clase tiene una sola instancia y provee un punto global para accesarla.
- **Aplicabilidad:** cuando debe de haber una sola instancia de una clase y debe ser accesible a los clientes desde un punto de acceso conocido.
- **Estructura** (figura 4.21)

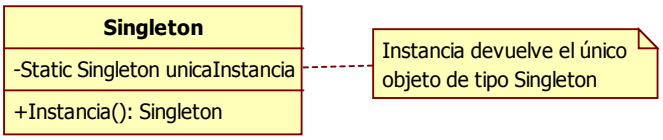


Figura 4.21: Patrón Singleton.

- **Participantes**
  - *Singleton:* define una operación Instancia que permite a los clientes tener acceso a su atributo *unicaInstancia*. Instancia es un método en una clase.

Por medio del patrón Singleton existirá una sola instancia que sirva como punto de acceso a las diferentes Fábricas concretas. Note que además de lo anterior, este patrón sirve como un medio común para tener acceso a diferentes clases, lo cual es el principio del patrón Fachada. Los diferentes subsistemas de la arquitectura dependerán de la Fachada/Singleton para crear sus objetos; ver figura 4.22.

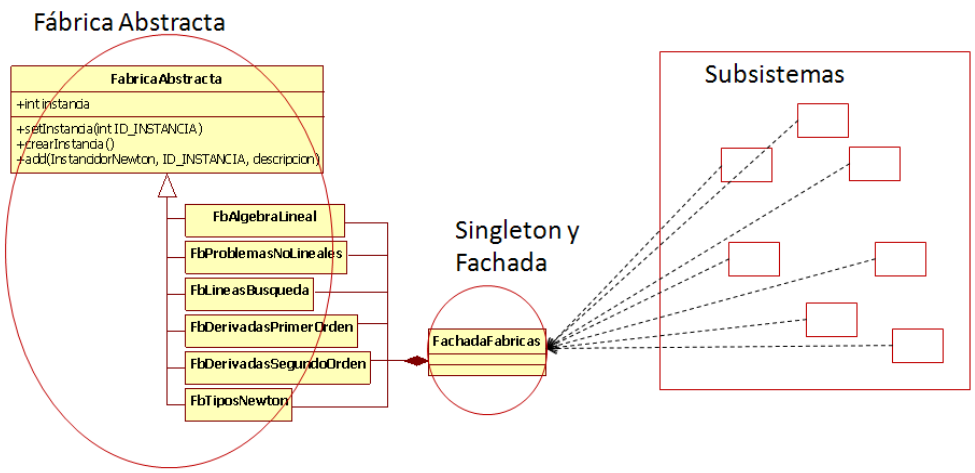


Figura 4.22: Diseño para la creación de objetos.

4.3.5. Interacción con paquetes externos

La interacción entre la arquitectura y los paquetes externos es un tema delicado. La arquitectura debe de tener la capacidad para intercambiar este tipo de paquetes sin que se altere su estructura interna ni se generen fallas de regresión. Por ejemplo, la arquitectura debería interactuar con paquetes de álgebra lineal ya que los algoritmos necesitan realizar muchas operaciones con vectores y matrices; si el paquete de álgebra lineal está acoplado, tratar de cambiarlo involucraría hacer cambios en prácticamente toda la arquitectura.

Para desacoplar los paquetes externos la arquitectura debe de tener una interfaz propia y el paquete externo debe adaptarse a esta interfaz. Esto se logra por medio del patrón Adaptador y el patrón Fábrica Abstracta; ver figura 4.23. Por un lado, el patrón adaptador provee una interfaz que entiende el cliente (la arquitectura) y que lo desacopla del adaptado. Por otro lado, el patrón Fábrica Abstracta oculta la implementación del paquete externo instanciando los objetos de dicho paquete sin que la arquitectura lo tenga presente.

4.3.6. Evitando objetos mal cocinados

La primera tarea del software que implemente la arquitectura propuesta en esta tesis, será crear los objetos que intervienen en la misma y asegurarse que las referencias a los objetos estén debidamente inicializadas. Esta tarea puede

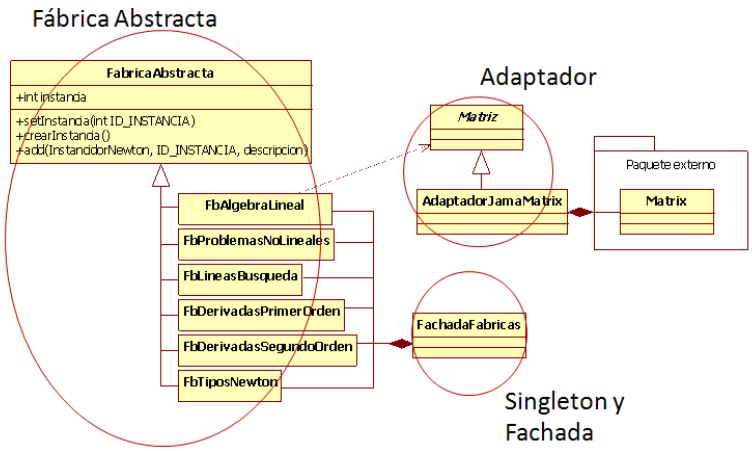


Figura 4.23: Diseño para interactuar con paquetes de externos.

complicarse debido a la interacción compleja entre los objetos. En la figura 4.24 se detalla la manera de crear los objetos con más interacción en la arquitectura.

4.4. Resumen de patrones

4.4.1. Descripción de patrones

En esta sección se resumen las ventajas que aportó cada patrón dentro de la arquitectura.

Puente

- Separa la abstracción de la implementación y permite ver a los métodos como una caja negra.
- La abstracción en este patrón representa a un método iterativo, lo que permite incorporar distintos métodos que no necesariamente son del tipo Newton.

Plantilla

- Provee un medio para que los distintos métodos Newton compartan su algoritmo genérico.

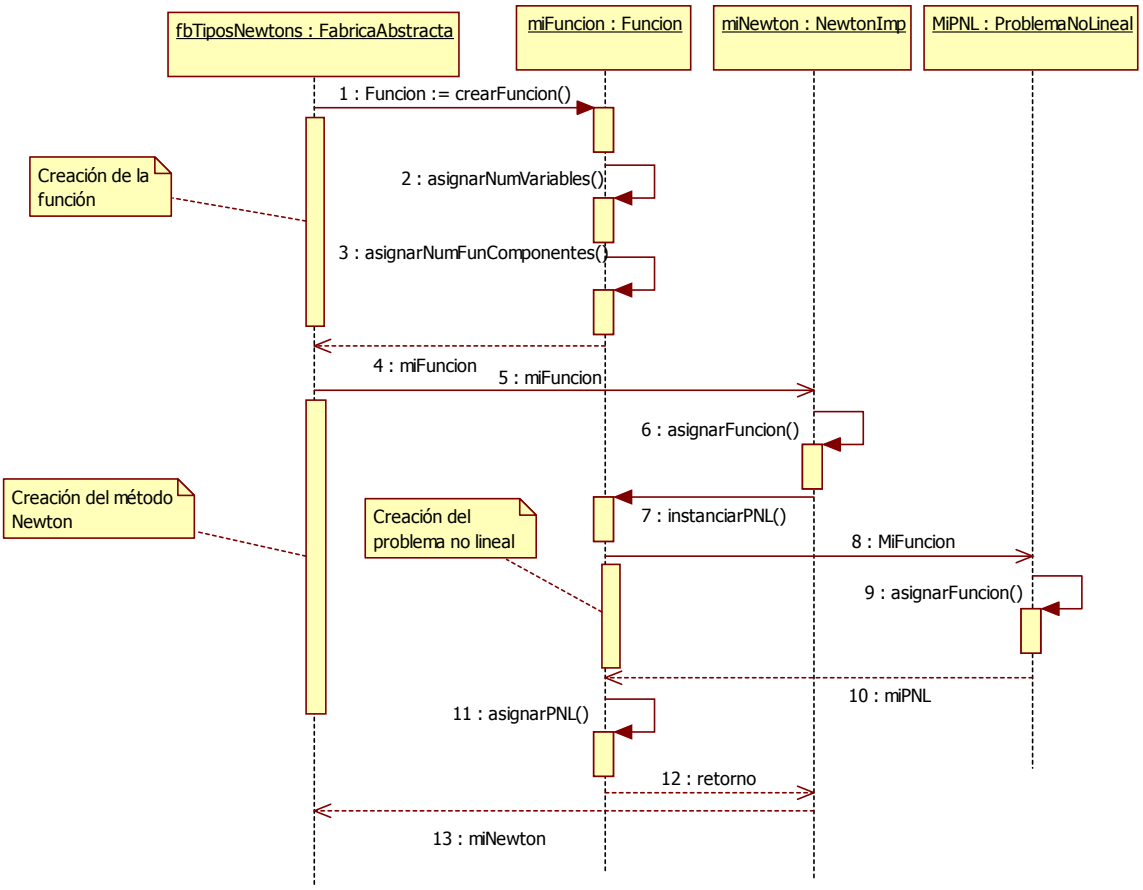


Figura 4.24: Creación de objetos y asignación de referencias entre los mismos.

- Permite variar las formas para calcular la longitud de paso, dirección Newton y criterio de paro, reutilizando el algoritmo en común que comparten los métodos Newton.

Fachada

- Encapsula las interfaces hacia las variantes del método Newton en un solo objeto, de tal manera que se facilite su manejo.

- Permite que la arquitectura base se comunique con los demás subsistemas de forma organizada.

- Provee una interfaz unificada hacia las distintas Fábricas concretas, lo que facilita su acceso para la creación de objetos.

Estado

- Oculta detalles de las derivadas para funciones con distintas formas, lo que permite un código más claro.

- Facilita la construcción del modelo lineal y cuadrático en los que se basan los métodos Newton.

- Facilita el intercambio entre problemas no lineales, lo que facilita la implementación de métodos Newton para diversas aplicaciones.

Fábrica abstracta

- Define una interfaz para la creación de cada miembro de las distintas familias de objetos involucradas en la arquitectura.

- Oculta la implementación de paquetes externos, y con ayuda del patrón Adaptador desacopla dichas paqueterías de tal forma que puedan ser reemplazadas cuando sea necesario.

- Permite indicar qué miembro de una familia de objetos debe ser instanciado, con lo cual se pueden crear diferentes instancias de la arquitectura.

Adaptador

- Permite adaptar las direcciones Newton al problema con restricciones de las regiones de confianza.
- Adapta la interfaz de los paquetes externos a una que sea conocida por la arquitectura.

Estrategia

- Facilita el intercambio de algoritmos para buscar posibles longitudes de paso en líneas de búsqueda.
- Permite variar distintas condiciones de decremento en líneas de búsqueda
- Permite variar distintos algoritmos de actualización de radio de confianza en regiones de confianza.

Singleton

- Proporciona un punto de acceso global que facilita el acceso a las distintas Fábricas concretas.

4.4.2. Vista general de la arquitectura y los patrones

En esta sección se presenta un diagrama en donde se muestra la arquitectura de forma global y la ubicación de cada patrón dentro de la misma; ver figura 4.25.

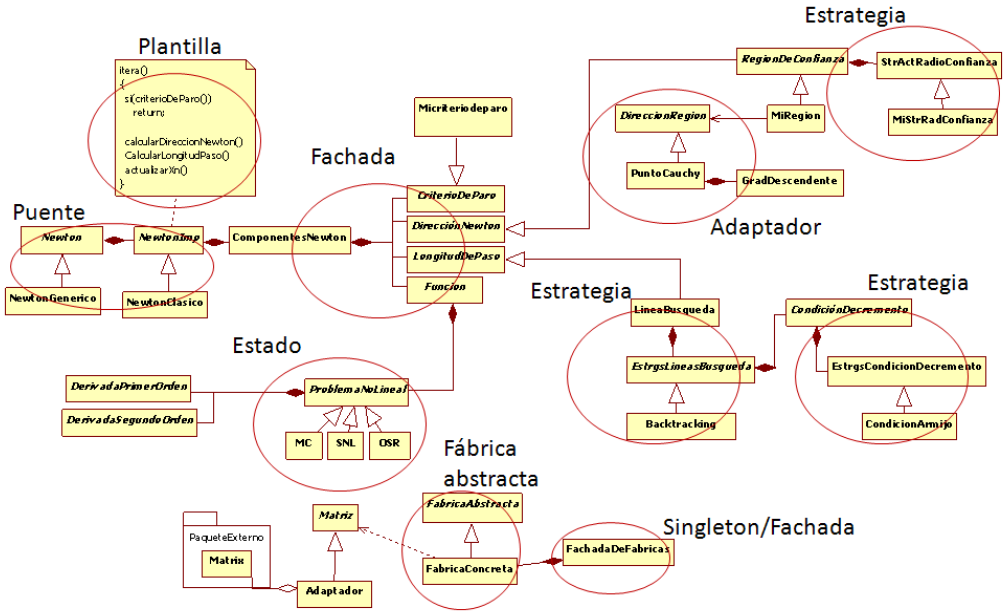


Figura 4.25: Vista general de la arquitectura y ubicación de los patrones.

Capítulo 5

Compromisos en el diseño

Durante el diseño arquitectónico, el diseñador tiene que tomar decisiones acerca de aspectos clave que afectan la construcción del sistema. Generalmente estas decisiones traen ventajas y consecuencias [Som04]. En este capítulo, se mencionan los principales compromisos involucrados en la arquitectura y las consecuencias de usar las soluciones de diseño propuestas en esta tesis.

El diseño arquitectónico involucra diferentes compromisos que son tratados dependiendo del propósito del sistema. Por ejemplo, algunas arquitecturas usan el concepto de redundancia para que sean especialmente tolerantes a los fallos, sin embargo, esto repercute en el desempeño de sistema [BMR+08]. En general, los sistemas pensados para facilitar el mantenimiento suelen tener menor desempeño [Som04]. Algunos otros sistemas tienen requerimientos grandes en cuanto a eficiencia y tratan de tener la menor cantidad de subsistemas para aminorar el costo de comunicación entre ellos, lo que puede traer como consecuencia software difícil de mantener.

En lo que se refiere a esta tesis, la organización de la arquitectura propuesta está basada en la descomposición del método, en diferentes subsistemas, tomando en cuenta la forma iterativa del mismo. Esto, por un lado, facilita la comunicación entre diversos métodos y el intercambio entre los diferentes componentes del método, pero por otro lado, eleva el costo de comunicación entre subsistemas. Podría decirse, que los subsistemas tienen una granularidad "fina" que permite características como flexibilidad y mantenimiento. Sin embargo la comunicación entre subsistemas puede ser más difícil de tratar.

En la arquitectura propuesta, la disponibilidad de información como el valor

de la función o las derivadas es necesaria a través de los diferentes subsistemas. También se necesita aminorar, en lo mayor posible, el número de llamadas a la función de evaluación. Por lo tanto, se requiere almacenar el valor de la función y sus derivadas en la iteración actual, para que así los subsistemas puedan disponer de ella. En ocasiones, algunos subsistemas necesitan volver a calcular la información mencionada pero en puntos diferentes al tratado en la iteración actual; toda esta información podría ser útil para los demás subsistemas y tiene que considerarse. Lograr la coordinación con el fin de tener la información necesaria y que ésta sea consistente con respecto a el propósito de cada subsistema es una tarea que se dificulta debido a la granularidad "fina" ya mencionada. Durante la implementación de la arquitectura este aspecto se trató centralizando la información de relevancia en un punto específico y cada subsistema recurriría a este punto para obtener la información necesaria, emulando así a un modelo cliente-servidor. Esta solución parece sensata, sin embargo, el diseñador que encuentre adecuado reutilizar las ideas propuestas en esta tesis debe tener en cuenta este compromiso.

En lo que respecta al patrón Estado, brinda buenas características para facilitar la implementación de métodos Newton en distintos problemas no lineales. Permite inclusive el cambio de problemas no lineales en tiempo de ejecución. Sin embargo, el uso desmesurado de cambios de estado puede traer inconsistencias en cuanto a las formas de las derivadas. Considere, por ejemplo, el caso de que la función se encuentre en un estado diferente al que espera el cliente, esto podría traer como consecuencia que el patrón brinde un gradiente cuando el cliente espera una matriz jacobiana o viceversa. Este tipo de situaciones pueden ser fácilmente identificables, y si el patrón se usa de forma correcta no deberían presentarse. Sin embargo, el programador que implemente el patrón debe considerar estos aspectos como situaciones excepcionales que deben tomarse en cuenta para lograr software robusto.

Los diversos subsistemas que conforman la arquitectura son altamente dependientes de las fábricas, ya que éstas brindan los objetos con los cuales los subsistemas realizarán sus operaciones. A su vez, las fábricas necesitan saber de las clases que conforman dichos subsistemas debido a que los objetos que crean son instancias de estas clases. Esta característica puede provocar ciclos entre los paquetes o subsistemas, lo que debe ser tomado en cuenta por el diseñador. Un cambio radical como agregar toda una nueva familia de objetos, es una decisión de diseño importante porque estos ciclos pueden provocar dependencias entre los

paquetes que tal vez no se vean a primera instancia. Al utilizar el patrón Fábrica, el diseñador tiene que tener en cuenta este aspecto y debe de ubicar las fábricas en lugares donde sean menos propensas a crear dependencias (Por ejemplo, en el paquete en donde se encuentren las clases que va a instanciar).

Otro aspecto de los patrones, es que utilizan principios como favorecer la composición a la herencia o delegar responsabilidades con la finalidad de lograr diseños para el cambio. Usar estos principios, aunado con la interacción entre los patrones, puede provocar diseños difíciles de entender para personas que estén poco familiarizadas con los patrones. El diseñador debe tener en cuenta que usar patrones con el propósito de lograr arquitecturas de software con buenas cualidades como flexibilidad, extensibilidad y fácil mantenimiento, puede provocar diseños complejos que sólo serán bien entendidos por las personas que conozcan dichos patrones. En este sentido, un aspecto favorable es que cada patrón tiene una idea intuitiva que describe el problema de diseño y la solución a dicho problema. Esta idea intuitiva puede ser enseñada a las personas involucradas en la construcción del software.

El uso de patrones permite crear buen software, pero reiteramos que su uso tiene un precio que tiene que ser evaluado por el diseñador de acuerdo al propósito del sistema que quiera construir. Esperamos que este capítulo, aunque breve, sea de ayuda para que las personas, que quieran reutilizar las soluciones de diseño expuestas en esta tesis, estén conscientes de los compromisos que conllevan las arquitecturas como la presentada. En la tabla 5.1 se presenta un resumen de los compromisos presentes en el diseño de la arquitectura.

Tabla 5.1: Compromisos de diseño presentes en la arquitectura

Decisión de diseño	Ventajas	Desventajas
Subsistemas con granularidad fina.	Mejor entendimiento del sistema y fácil mantenimiento.	La comunicación y coordinación entre subsistemas es más difícil de tratar.
Intercambio entre problemas no lineales mediante el patrón Estado.	Fácil manejo de derivadas y modelos basados en series de Taylor, para diferentes problemas no lineales. Capacidad para cambiar problemas no lineales en tiempo de ejecución.	El cambio de estados de forma desmesurada puede provocar inconsistencias en las formas de las derivadas o funciones.
Uso de fábricas para instanciar objetos.	Permite variar la instanciación de objetos independientemente de la lógica del programa.	Las Fábricas pueden provocar ciclos en el diseño debido a la alta interdependencia entre la Fábrica y los subsistemas.
Uso de patrones.	Diseños pensados para el cambio con las características de flexibilidad, extensibilidad y fácil mantenimiento.	Diseños complejos que sólo son bien entendidos por personas familiarizadas con los patrones.

Capítulo 6

Verificación y validación

Para evaluar el diseño se realizó la verificación y validación del mismo. Nos agrada la manera de Boehm para distinguir estos dos aspectos [Boe79]:

- Validación: ¿Se está construyendo el producto correcto?.
- Verificación: ¿Se está construyendo correctamente el producto?.

El objetivo de la validación es asegurarse que el producto satisface las expectativas del usuario, mientras que la verificación se refiere a construir el software de acuerdo a su especificación [Som04].

En este trabajo se validó que el diseño cubra con las expectativas de un posible usuario en un caso de estudio recurrente en la práctica, en donde se tienen que evaluar diferentes métodos de optimización para poder encontrar los parámetros adecuados en un sistema dinámico. Por otro lado, verificamos que el diseño esté de acuerdo con su grado de abstracción e inestabilidad y que sea propenso a extenderse sin abusar de su abstracción. Asimismo, se verifica que las clases interactúen debidamente y no caigan en la zona poco útil o la zona del dolor; ver figura 2.7.

6.1. Caso de estudio: estimación de parámetros para un motor de corriente continua

Esta sección está dividida en cuatro partes. En la primeras dos secciones, se describe en qué consiste el caso de estudio, y se dan algunas especificaciones acerca de la experimentación. Enseguida, se describen las necesidades del usuario

en cuanto al diseño. También se hace una pequeña comparación entre una familia de métodos en Fortran y la arquitectura propuesta, cuando éstos se utilizan para este tipo de pruebas. Para finalizar, se hacen algunas observaciones breves acerca de la comparación entre los diversos métodos.

6.1.1. Descripción del caso de estudio

Las tareas de optimización son recurrentes en el ámbito científico y de ingeniería. En el diseño de sistemas de control se necesitan métodos de optimización para encontrar las entradas de control adecuadas para que el sistema se comporte de forma eficiente. La dinámica del sistema se representa por medio de un sistema de ecuaciones diferenciales:

$$\dot{\mathbf{x}}(t) = \mathbf{F}(\mathbf{x}(t), u(t), t), \quad \mathbf{x}(t_0) = \mathbf{x}_{0,}$$

donde $\dot{\mathbf{x}} = d\mathbf{x}/dt$, $\mathbf{x} \in \mathbb{R}^n$ y u es un vector con las entradas de control. El tiempo inicial t_0 y el tiempo final t_f son dados. El desempeño del sistema durante el intervalo $[t_0, t_f]$ se mide por medio de la función costo

$$J(u) = \phi(\mathbf{x}(t_f, u)).$$

El problema consiste en encontrar una u que minimice la función costo. Primero se simula el sistema hasta el tiempo final y enseguida se evalúa la función costo. Este proceso se repite usando el algoritmo de optimización hasta encontrar los parámetros de control de entrada óptimos.

En este caso de estudio se tiene el modelo de un motor de corriente continua expresado con el sistema siguiente [Has76]:

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -\frac{B}{J_{mm}} & K_T \\ 0 & 0 & -\frac{R_f}{L_f} \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ 0 \\ \frac{1}{L_f} \end{bmatrix} u,$$

$$\mathbf{y} = [1, 0, 0]\, \mathbf{x}.$$

Las condiciones iniciales del sistema son $\mathbf{x}(t_0) = \mathbf{0}$, donde $x = [x_1, x_2, x_3]$, $x_1 = \theta$ es la posición del eje, $x_2 = \dot{\theta}$, $x_3 = i_f$ es la corriente que controla la torsión o *torque* del motor, R_f es la resistencia del circuito estator, L_f es la inductancia en

la bobina, J_{mm} es el momento de inercia en el eje, K_T es una constante, B es el amortiguamiento rotacional y $u = e_f$ es el voltaje aplicado.

La ley de control para el sistema del motor está dada por

$$u = -\mathbf{K}^T\mathbf{x} + k_1 r(t), \mathbf{K} = [k_1, k_2, k_3]$$

y r es el valor deseado para la posición angular del eje. Encontrar la u óptima requiere, por tanto, de encontrar el vector $\mathbf{K}$ que minimice la función costo.

La forma de la función costo es la siguiente

$$J(u) = \int_{t_0}^{t_f} [(r(t) - \theta(t))^2 + Ru^2(t)]dt,$$

donde R es una constante para limitar la energía aplicada.

Los valores de los parámetros durante la experimentación están en la tabla 6.1.

Tabla 6.1: Parámetros usados en el caso de estudio.

B = 2.0/3.0
Jmm = 2.0
KT = 10.0
Lf = 10.0
Rf = 10.0
R = 0.001
r = 10 en el intervalo $[t_0, t_f]$
$K_0 = [10, 10, 10]$

La intención de las pruebas usando este caso de estudio es: i) validar si el diseño cumple con las necesidades de diseño de un posible usuario, ii) evaluar la convergencia de los distintos métodos de optimización en cuanto a la calidad de la dirección utilizada y la longitud de paso que brinda la línea de búsqueda. Primero se probaron los métodos usando la línea de búsqueda correspondiente, luego se omitió este aspecto con la intención de medir la calidad de la dirección usada.

6.1.2. Descripción de los métodos de optimización

Se implementaron distintos métodos del tipo Newton y gradiente descendente; así como una línea de búsqueda bastante sencilla que utiliza una técnica de bisección. Éstos se describen a continuación:

1. El *gradiente descendente* usa la dirección $\mathbf{d} = -\boldsymbol{\nabla} f$ y una longitud de paso $\lambda = 1$.

2. El método *Newton Armijo hessiana diferencias* utiliza la dirección $\mathbf{H}f\ \mathbf{d} = -\boldsymbol{\nabla}\mathbf{f}$, donde $\mathbf{H}f$ es una aproximación por diferencias finitas y usa una estrategia de línea de búsqueda que prueba la sucesión $1, \frac{1}{2}, \frac{1}{4}, ..., \frac{1}{2^n}$.

3. El método *Newton armijo BFGS* es igual al método anterior pero $\mathbf{H}f$ es una aproximación quasi-Newton de la matriz hessiana dada por las fórmulas (6.1) y (6.2).

$$\mathbf{H}f_{k+1} = \mathbf{H}f_k - \frac{\mathbf{H}f_k\mathbf{s}_k\mathbf{s}_k^T\mathbf{H}f_k}{\mathbf{s}_k^T\mathbf{H}f_k\mathbf{s}_k} + \frac{\mathbf{y}_k\mathbf{y}_k^T}{\mathbf{y}_k^T\mathbf{s}_k}, \tag{6.1}$$

$$\mathbf{s}_k = \mathbf{x}_{k+1} - \mathbf{x}_k\ ,\ \ \mathbf{y}_k = \nabla f_{k+1} - \nabla f_k \tag{6.2}$$

Los métodos anteriores se compararon con una familia de métodos de gradiente conjugado: *Fletcher Reeves, Polak Ribiere y Positive Polak Ribiere*. La descripción de éstos, así como la línea de búsqueda sofisticada que utilizan se encuentra en el trabajo de Gilbert y Nocedal [Gh92].

6.1.3. Requerimientos del caso de estudio

A continuación se describen las necesidades de diseño del usuario y cómo éstas son cubiertas por la arquitectura propuesta.

1. **Interfaz para incorporar la función costo**: esta interfaz es parte de la arquitectura base; ver sección 4.1.6.

2. **Capacidad para interactuar con varios integradores para resolver el modelo del motor:** para cubrir este requerimiento se necesita que la arquitectura use paquetes especializados en técnicas de integración. Esto

puede ser tratado por medio de un patrón Adaptador y la Fábrica Abstracta, como se ejemplificó en la sección 4.3.5.

3. **Capacidad para intercambiar diferentes métodos de optimización**: este servicio es brindado por el patrón Puente, ya que permite variar las distintas implementaciones de métodos de optimización que tengan la forma iterativa especificada en la sección 4.1.

4. **Capacidad para incorporar distintas técnicas para calcular derivadas**: esto es cubierto mediante la familia de clases especificada en la sección 4.3.3.

5. **Capacidad para incorporar distintos tipos de líneas de búsqueda**: el diseño brinda la flexibilidad para implementar distintas estrategias para buscar las longitudes de paso y, a su vez, variar las distintas condiciones de decremento. Esto permite integrar un amplio conjunto de técnicas de este tipo; ver sección 4.3.1.

La arquitectura propuesta facilita la comparación entre métodos ya que provee una interfaz común para los métodos que compartan la forma iterativa plasmada en la arquitectura base. Además, realizar un cambio sencillo como omitir la longitud de paso en las rutinas en Fortran utilizadas para llevar a cabo las comparaciones presentadas [Gh92], requiere de: i) rastrear la variable referente a la longitud de paso en distintas subrutinas, ii) tener que modificar el código existente en las subrutinas iii) compilar el código modificado.

Usando la arquitectura propuesta se puede crear una clase que extienda *EstrgLineaBusqueda*; ver figura 4.15. Dicha clase puede implementar un algoritmo de línea de búsqueda "burdo" que retorne una longitud de paso de uno. Con esto, se extiende el código pero no se modifica el código existente. Además por medio del patrón Estrategia se pueden hacer cambios, en tiempo de ejecución, entre esta línea "burda" y otra línea. Por tanto, esta clase de pruebas se puede realizar de forma muy sencilla.

6.1.4. Evaluación de los métodos de optimización

El software que facilite la comparación de métodos es necesario ya que da pauta a la toma de decisiones. En este caso queremos evaluar qué tan buenos son los

métodos con su línea de búsqueda (figura 6.1) y sin ella (figura 6.2,). Por ejemplo, en la figura 6.1, se puede ver que los métodos de gradiente conjugado (*Fletcher Reeves*, *Polak Ribiere* y *Positive Polak Ribiere*) tienen una mejor convergencia que los métodos del tipo Newton. Esto es peculiar ya que es bien sabido que los métodos Newton tienen mejor convergencia que los de gradiente conjugado. En la figura 6.2, se aprecia que la razón de lo anterior se debe a la línea de búsqueda utilizada por los métodos de gradiente conjugado. También se puede notar que el *Newton Armijo hessiana diferencias* diverge, lo que quiere decir que la implementación de diferencias finitas para aproximar la matriz hessiana puede no ser tan buena.

Por otro lado, el *Newton armijo BFGS* tiene una convergencia muy rápida a pesar de no usar ninguna estrategia de convergencia global. Esto se debe a que la primera estimación de la matriz hessiana usada en este método es la misma que la de *Newton Armijo hessiana diferencias*. Sin embargo, las fórmulas (6.1) y (6.2) permiten mejorar la aproximación de la matriz hessiana en cada iteración. De ser necesario un método aún mejor que los mostrados, el usuario puede considerar usar la dirección de *Newton armijo BFGS* y utilizar una línea de búsqueda más sofisticada como la usada con los algoritmos de gradiente conjugado.

Si se toma $R = 0{,}001$ se puede calcular el valor "exacto" $K = [31.6228, 22.3485, 38.3203]$ [Kir04], y con esto estimar el error absoluto $\|K - K^*\|$ y el error relativo $\frac{\|K-K^*\|}{\|K\|}$, donde K^* es el resultado dado por los métodos evaluados. La norma usada en este caso es la euclidiana, los errores de los métodos con y sin línea de búsqueda están en la tabla 6.2 y la tabla 6.3, respectivamente.

6.2. Midiendo el diseño

Durante el desarrollo de esta tesis se implementó el diseño de la arquitectura propuesta y se hicieron diversas pruebas con sistemas de ecuaciones no lineales y optimización sin restricciones. Por ejemplo, se hicieron algunas implementaciones de Newton amortiguado que aparecen en la literatura [CB80, Kel03] y algunos métodos híbridos que usan regiones de confianza [Pow70, Blu80, WhN99]. Esto, junto con el caso de estudio presentado y a pesar de que las implementaciones realizadas no son las mejores, permitió que la arquitectura creciera, evaluando así, si el diseño presentaba buenas cualidades en cuanto a flexibilidad y extensibilidad.

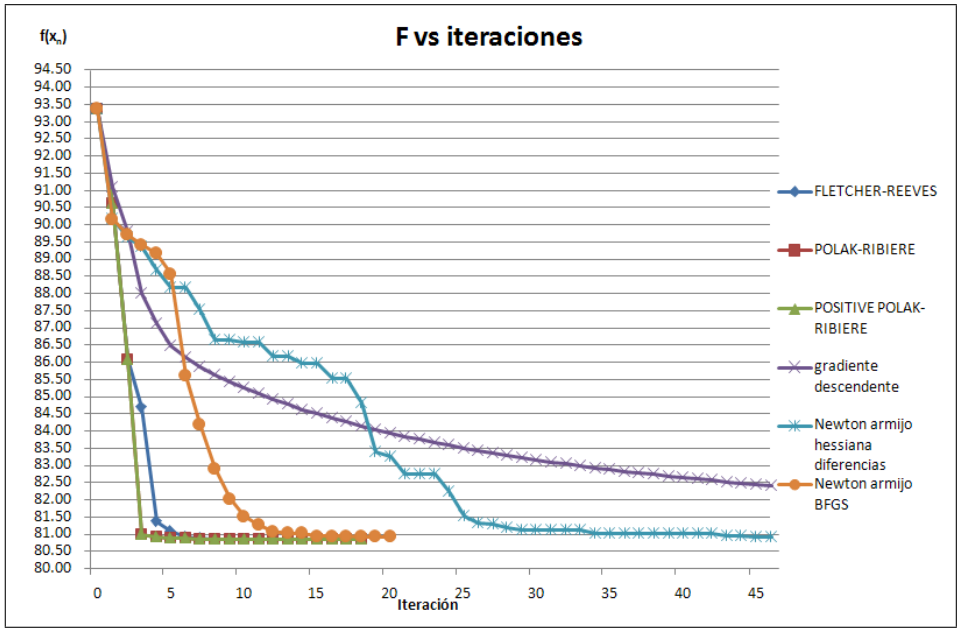


Figura 6.1: Convergencia de los métodos usando línea de búsqueda.

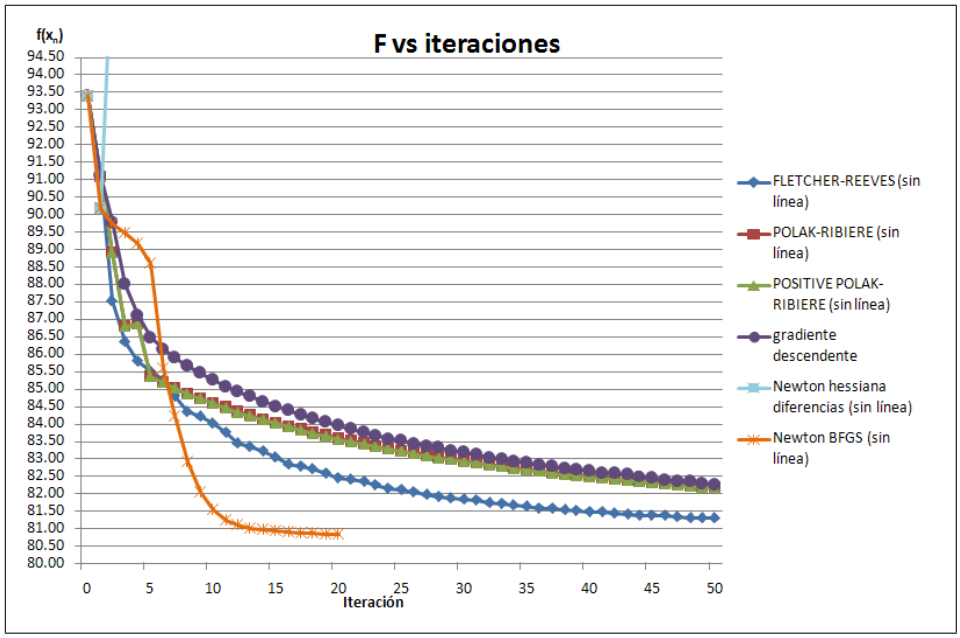


Figura 6.2: Convergencia de los métodos sin usar línea de búsqueda.

Tabla 6.2: Error absoluto y relativo de los métodos usando línea de búsqueda

Método	**Iteraciones**	$\mathbf{k_1}$	$\mathbf{k_2}$	$\mathbf{k_3}$	**Error absoluto**	**Error relativo**
FLETCHER REEVES	16	31.5564	22.2953	38.2294	0.1243	0.0022
POLAK RIBIERE	18	31.5565	22.2953	38.2295	0.1242	0.0022
POSITIVE POLAK-RIBIERE	18	31.5565	22.2953	38.2295	0.1242	0.0022
gradiente descendente	200	28.0243	20.5350	31.1835	8.1958	0.1504
Newton Armijo hessiana diferencias	46	31.1169	21.1919	38.0887	1.2833	0.0235
Newton Armijo BFGS	20	32.3711	23.4013	41.0777	3.0450	0.0558

Tabla 6.3: Error absoluto y relativo de los métodos sin usar línea de búsqueda

Método	**Iteraciones**	$\mathbf{k_1}$	$\mathbf{k_2}$	$\mathbf{k_3}$	**Error absoluto**	**Error relativo**
FLETCHER REEVES	200	30.6913	21.8700	36.4646	2.1307	0.0391
POLAK RIBIERE	200	28.0741	20.5840	31.3100	8.0528	0.1478
POSITIVE POLAK-RIBIERE	200	28.1067	20.6000	31.3725	7.9806	0.1464
gradiente descendente	200	28.0243	20.5350	31.1835	8.1958	0.1504
Newton Armijo hessiana diferencias	-	-	-	-	-	-
Newton Armijo BFGS	20	31.7578	22.2194	38.9130	0.6214	0.0114

Se usaron las métricas de Robert Martin [Mar03] para medir la arquitectura (estas métricas se explican en la sección 2.2.6 del marco teórico). Medimos la abstracción e inestabilidad de cada paquete conforme crecía el software que implementa el diseño propuesto. Una descripción breve de las clases y su distribución en cada paquete está en el apéndice A. En esta sección sólo se muestran las gráficas de los resultados obtenidos al medir la arquitectura, puede consultar estos resultados a detalle en el apéndice B.

Al medir los paquetes se buscaron dos aspectos principalmente: i) asegurarse de que la arquitectura base soporte el crecimiento del software verificando que cumpla con un nivel de abstracción e inestabilidad adecuado mientras más paquetes interactúan con ella y ii) identificar paquetes cercanos a la zona de poco uso y la zona del dolor. En las figuras 6.3, 6.4, 6.5 y 6.6, se presenta la abstracción e inestabilidad del prototipo que implementa la arquitectura. Medimos dichos aspectos conforme el software crecía, para así poder apreciar si la arquitectura base se mantenía cerca de la secuencia principal aún cuando más paquetes interactuan con ella; asi mismo, nuestra intención era identificar paquetes que comenzarán a acercarse a las zonas no deseadas.

En la figura 6.3 se presenta el prototipo de la arquitectura cuando sólo tenía cinco paquetes. Los paquetes tienen una abstracción e inestabilidad ideal. Por un lado, hay paquetes muy abstractos y estables; por otro lado, hay paquetes inestables pero poco abstractos. El paquete *arquitecturabase* es muy abstracto y estable, además pasa por la secuencia principal. En el paquete *arquitecturabase* se encuentran las interfaces de las cuales dependerán paquetes inestables como *criteriosparo*, *direcciones*, *funciones* y *lineasbusqueda*. Los paquetes pueden depender de *arquitecturabase* de forma segura, ya que su estabilidad es una garantía de que ese paquete no será propenso a cambiar demasiado.

En la figura 6.4 se presenta el prototipo cuando contaba con trece paquetes. En esta instancia el prototipo tiene incorporado el diseño concerniente a la creación de objetos (paquete *fabricaabstracta*), el diseño para los métodos Newton que usan líneas de búsqueda (*lineasbusqueda*), el diseño para adaptar paquetes de algebra lineal (*alebralineal*), etc. El paquete *arquitecturabase* tiende a ser un poco más inestable, sin embargo continúa cerca de la secuencia principal. Nos agrada que la mayoría de los paquetes se encuentran "cerca" de la secuencia principal. Una excepción a lo anterior es el paquete *precisionnumerica.*

En la figura 6.5 se ha agregado el diseño de regiones de confianza y el prototipo

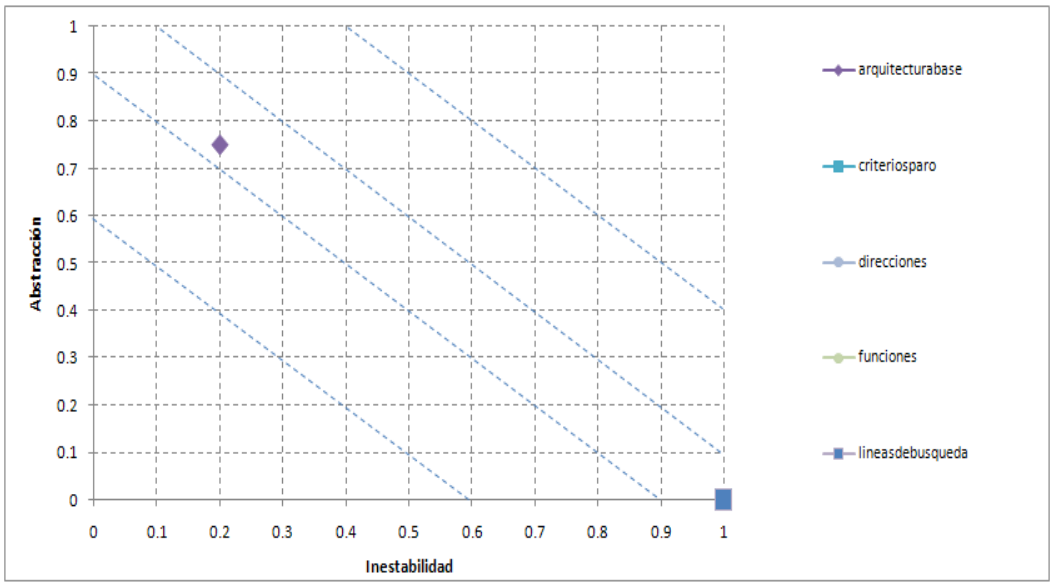


Figura 6.3: Inestabilidad y abstracción de la arquitectura base y cuatro paquetes principales. Notará que los paquetes *direccion, funciones, criteriosparo* y *lineasbusqueda* están traslapados en (1,0). Puede consultar una descripción detallada de estos resultados en el apéndice B.

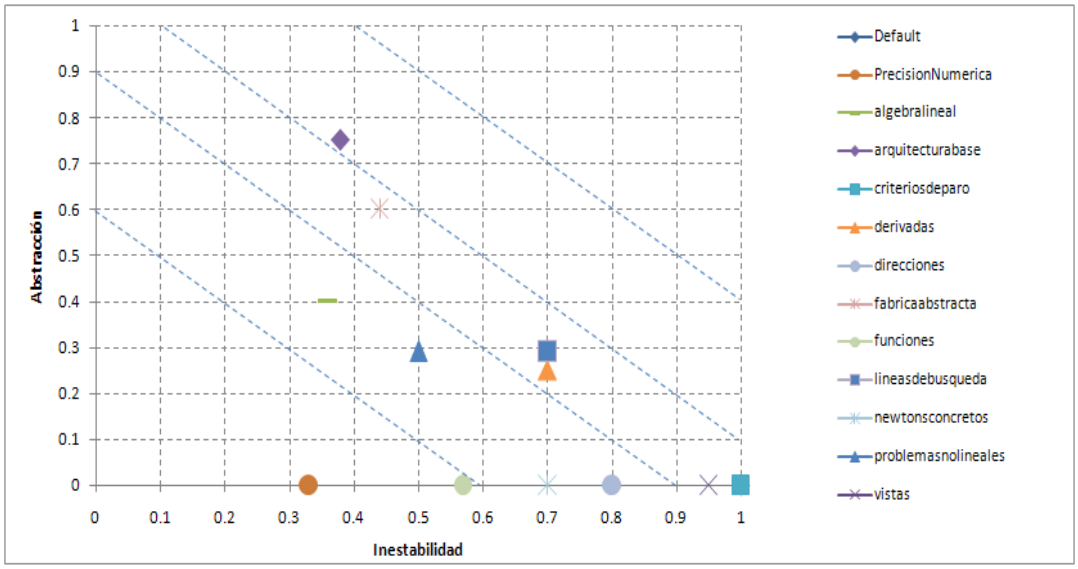


Figura 6.4: Inestabilidad y abstracción de los paquetes tomando en cuenta el diseño de las líneas de búsqueda.

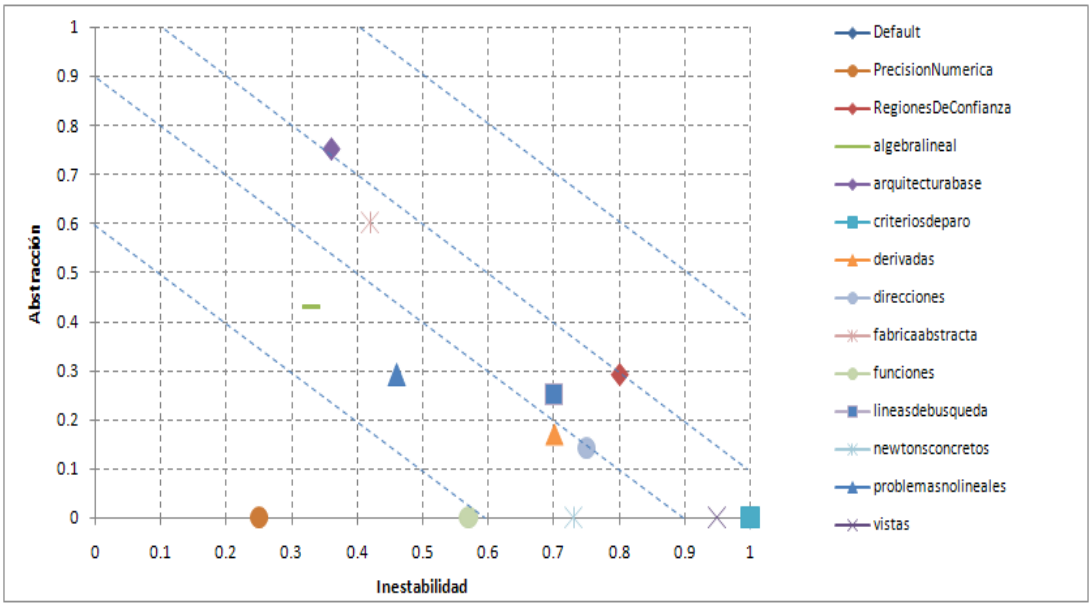


Figura 6.5: Inestabilidad y abstracción de los paquetes al agregar el diseño de lineas de búsqueda y regiones de confianza.

cuenta con catorce paquetes. Podemos notar que el paquete *arquitecturabase* continúa en la secuencia principal. La mayoría de paquetes permanecen inmóviles aún cuando *regionesconfianzaDeConfianza* es agregado al prototipo. Esta característica es favorable porque indica que la arquitectura puede crecer sin la necesidad de modificar la mayoría de los paquetes.

Por último en la figura 6.6 se muestra el estado del prototipo hasta la escritura de esta tesis (que por el momento tiene una cantidad de 17 paquetes). En esta instancia fue incorporado el caso de estudio y algunos paquetes para calcular derivadas por medio de diferencias finitas. El paquete arquitecturabase se mantiene las cualidades previamente mencionadas. Otro aspecto a tomar en cuenta, es que la mayoría de los paquetes más abstractos no se mueven demasiado y permanecen cerca de la secuencia principal. El mayor movimiento se puede notar en los paquetes menos abstractos. Esta particularidad es consistente con el principio de inestabilidad y abstracción, ya que los paquetes más abstractos deben ser menos propensos a modificarse (estables) y los paquetes menos abstractos deben ser mas volátiles (inestables). Algunas excepciones a este principio fueron los paquetes *precisionnumerica* y *JacobianMatriz.Abstraccion*. A continuación se describe qué es lo que sucede con estos paquetes.

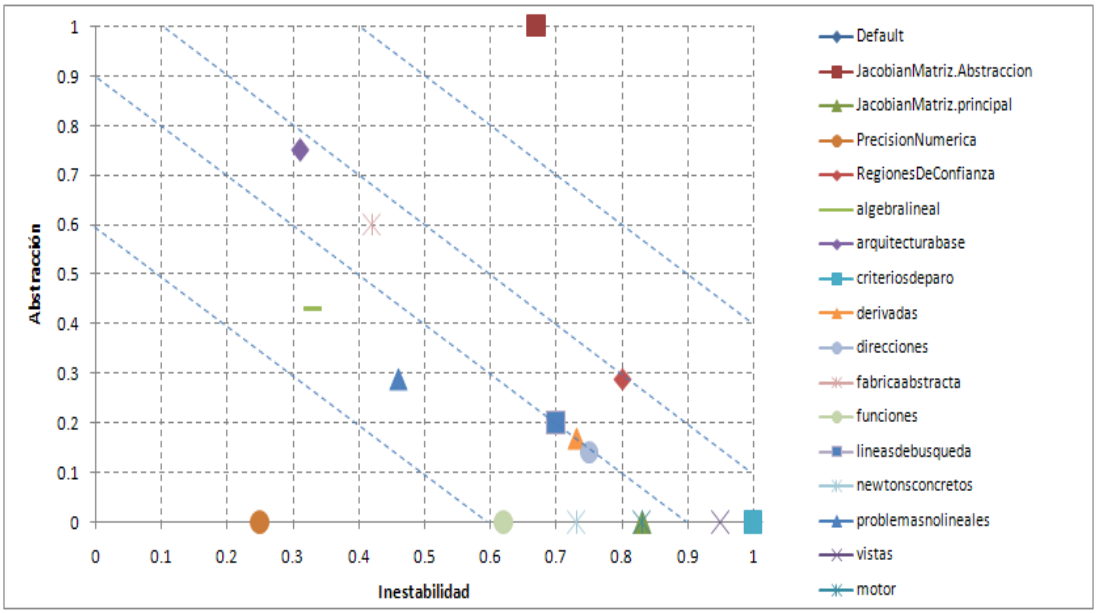


Figura 6.6: Inestabilidad y abstracción de los paquetes tomando en cuenta líneas de búsqueda, regiones de confianza y el caso de estudio.

El paquete *precisionnumerica* está cerca de la zona del dolor. Esto es debido, en parte, a que diseñar este paquete estaba fuera del alcance de los objetivos en esta tesis. Sin embargo, este paquete calcula aspectos relacionados con la representación de punto flotante de la máquina lo cual es necesario para la implementación de los métodos. Se requiere que este paquete presente la suficiente abstracción para extenderlo, así el usuario podrá incorporar las variables de representación de punto flotante de su máquina.

El paquete *JacobianMatriz.Abstracción* es demasiado inestable y abstracto. Esto quiere decir que no es seguro extenderlo. El paquete fue incorporado durante el caso de estudio con el objetivo de utilizar una estrategia de diferencias finitas que se encuentra en él. Lo mejor es rediseñar este paquete. La incorporación de este paquete fue realizada por medio de un patrón Adaptador y por tanto puede ser reemplazado sin dificultades.

Nos agrada que el paquete que contiene la arquitectura base, se mantenga cerca de la secuencia principal a lo largo del crecimiento del software. Una buena característica de la arquitectura base es que su inestabilidad no crece conforme más paquetes interactúan con ella. Esto quiere decir que su estabilidad es una garantía para extender clases de sus interfaces.

Capítulo 7

Conclusiones

En esta tesis se realizó el análisis y diseño de una arquitectura que pretende solventar el problema de reuso en métodos del tipo Newton para optimización y sistemas de ecuaciones no lineales. Debido a la falta de experiencia de diseño de software en el área de cómputo científico, se retomó la experiencia del software de negocios, en forma de patrones de software, para construir la arquitectura. El principal problema del enfoque anterior, es el mapeo de los patrones hacia el área de cómputo científico. Durante el trabajo de tesis, este inconveniente se resolvió por medio un análisis que permite encontrar abstracciones adecuadas para entender un conjunto de variaciones como un todo. A continuación se dan las conclusiones del trabajo realizado en cuanto al análisis y el diseño de la arquitectura.

7.1. Acerca del análisis

El diseño está guiado por un análisis que permite identificar posibles familias de clases que compartan una interfaz común. Esto por medio del estudio de las variaciones de conceptos en distintos escenarios. El uso de este análisis, para guiar el proceso de diseño, no se muestra en los trabajos mencionados en el estado del arte [MCH+04, PSS09, PSS04, San09]. Durante la tesis, dicho análisis trajo las siguientes ventajas:

- La identificación de posibles participantes que interactúan en un patrón. Por ejemplo, en el patrón Estado se identificó el *contexto* y el *comportamiento* de la función de evaluación para cada *estado*.

- La identificación de conceptos que varían en la arquitectura base, en los algoritmos de líneas búsqueda y regiones de confianza. Esto permite saber cuáles son las partes del diseño que deben ser flexibles y cuáles deben ser estables.

- La identificación del criterio para construir las interfaces que comunican las distintas formas de calcular las derivadas.

- La identificación de los objetos que tienen que ser creados por las Fábricas. Al identificar cada familia de objetos, una fábrica concreta tendrá que instanciar cada miembro de la familia.

A manera de opinión personal, el análisis de variabilidad y partes comunes es una técnica bastante útil para la construcción de software para métodos o problemas bien estudiados en el área de cómputo científico. Esto porque muchos de los conceptos comunes coinciden con los conceptos teóricos del problema/método. En ocasiones, la teoría brinda las formas generalizadas compartidas por cierto tipo de métodos o problemas, con las cuales se pueden construir las abstracciones adecuadas en el software. En el caso de los métodos Newton, lo anterior es visible en la forma iterativa del método, la cual es el concepto en común en el que se sustenta la arquitectura base propuesta en esta tesis. Es verdad que muchos aspectos que se tienen que tomar en cuenta al construir el software no serán dados, ni considerados en la teoría. Ante esta situación, el ingeniero (a) de software puede considerar el análisis de variabilidad y partes comunes como una guía que lo lleve a soluciones novedosas o a la identificación de patrones que resuelvan su problema de diseño.

7.2. Acerca del diseño

La arquitectura propuesta está pensada para proveer los siguientes aspectos:

- Intercambio entre distintos métodos que compartan la forma iterativa de los métodos Newton.

- Facilitar los cambios en las diversas maneras de calcular la longitud de paso y dirección Newton.

- Variar los algoritmos para encontrar posibles longitudes de paso e intercambiar las condiciones de decremento, en los algoritmos de líneas de búsqueda.
- Intercambio en la concordancia, el algoritmo para actualizar el radio de confianza y el algoritmo para el problema con restricciones; en los algoritmos de regiones de confianza.
- Capacidad para utilizar la arquitectura para diversos problemas no lineales.
- Facilidad para incorporar distintas estrategias para calcular derivadas.
- Capacidad para interactuar con paquetes externos sin acoplarlos a la arquitectura.

Nuestra arquitectura está conformada por patrones de diseño y patrones de arquitectura usados comúnmente en el software de negocios o administrativo. La principal dificultad del enfoque de patrones es identificar problemas de diseño y los posibles patrones para resolver estos problemas. En esta tesis se logró retomar la experiencia del software administrativo, al identificar dichos patrones para resolver problemas relacionados con la familia de métodos tipo Newton. Dentro de los patrones identificados, el patrón Estado es útil para ocultar las formas de las derivadas para distintas funciones, además de facilitar la construcción de modelos basados en la serie de Taylor. Los paquetes que involucren la diferenciación de funciones de distintas formas pueden considerar este patrón dentro de su diseño.

Encontramos útil el diseño en capas para tratar a un método con tantas variaciones. Los distintos grados de abstracción en la arquitectura permiten manipular los métodos dependiendo de las necesidades del usuario. Un usuario inexperto puede hacer uso de la capa 4 para intercambiar los métodos a su disposición de manera sencilla. Un usuario experto puede hacer uso de la capa 1 para hacer modificaciones en partes específicas de un método. Las capas 2 y 3 se encargan de controlar la interacción entre las distintas variantes del método.

A pesar de que la arquitectura en un principio fue pensada para métodos del tipo Newton, muchas de las ideas de diseño plasmadas en la arquitectura pueden reusarse para diseñar software para otro tipo de métodos o problemas. El patrón Estrategia permite intercambiar algoritmos fácilmente, el patrón Plantilla permite variar los pasos en un algoritmo, mientras que la Fábrica Abstracta permite separar la creación de objetos de la parte lógica de los algoritmos. Estas ideas

pueden ser reusadas de forma separada para diferentes propósitos. Por otro lado, también se pueden retomar ideas acerca de cómo usar los patrones en conjunto. La arquitectura base es un buen ejemplo de cómo pueden interactuar los patrones para crear diseños en capas. La idea de "apilar" dos patrones Estrategia en las líneas de búsqueda, puede ser de ayuda cuando se quiera controlar la variación de dos algoritmos y uno de ellos sea fácil de manipular (no sea propenso a crecer demasiado o no dependa de demasiadas entradas para su funcionamiento).

La forma de evaluar el diseño fue medir el grado de abstracción e inestabilidad de los paquetes mientras el software, que implementa el diseño, crecía. Los resultados indican que es seguro utilizar la arquitectura base ya que es lo suficientemente estable. Durante la incorporación de algunos métodos Newton y el caso de estudio, fue necesario agregar paquetes para tratar los aspectos de representación de punto flotante y estrategias para calcular derivadas. Las métricas mostraron que dos de estos paquetes estaban mal diseñados. Sin embargo, el uso de patrones permite reemplazar o rediseñar estos paquetes sin alterar las demás partes de la arquitectura.

Se utilizó un caso de estudio en donde se tratan de evaluar distintos métodos de optimización con el propósito de encontrar los parámetros de un modelo de un motor de corriente continua. La arquitectura propuesta cumple con las posibles necesidades que el usuario necesitará en ese caso de estudio.

7.3. Aportes

Los siguientes son los principales aportes de esta tesis:

- El diseño de una arquitectura que tiene la característica de controlar las variaciones en diversos métodos Newton, y que está diseñada para crecer de forma segura.

- El uso de un análisis, novedoso en el área de cómputo científico, que sirve como guía para la identificación de patrones.

7.4. Trabajo futuro y posibles líneas de investigación

En cuanto a la arquitectura, se tiene que realizar diseño para tratar los criterios de paro. Este es un problema interesante que ocupa de diseño flexible para variar las normas vectoriales así como de intercambiar entre las distintas condiciones de paro. Por otro lado, la identificación de patrones en cómputo científico es una área de investigación que aún continúa abierta y que en los últimos años tenido más atención por parte de la comunidad. Lo anterior puede ser realizado de dos formas:

1. Identificando patrones propuestos para software de negocios o administrativo (como en el caso de esta tesis) que puedan ser utilizados para cómputo científico.

2. El estudio de diseño en las paqueterías de software para cómputo científico que han tenido éxito, con la finalidad de encontrar buenas soluciones que puedan ser propuestas como patrones.

Otro aspecto importante es el estudio de posibles metodologías que sirvan como guías para identificar patrones (existentes) que resuelvan problemas de diseño en el área de cómputo científico. Otro punto interesante es el estudio de los patrones encontrados hasta el momento, en el área de cómputo científico, con la finalidad de formar un lenguaje de patrones que ayude a describir los problemas de diseño en el área y las posibles soluciones para ellos.

Bibliografía

[Ale79] C. Alexander, *The timeless way of building*, Oxford University Press, USA, 1979.

[Ben00] E.A. Bender, *An introduction to mathematical modeling*, Dover Pubns, 2000.

[BGMS95] S. Balay, W.D. Gropp, L.C. McInnes, and B.F. Smith, *Petsc 2.0 users manual*, Argonne National Laboratory (1995).

[Bli02] C. Blilie, *Patterns in scientific software: An introduction*, Computing in Science and Engineering (2002), 48–53.

[Blu80] James L. Blue, *Robust methods for solving systems of nonlinear equations*, SIAM Journal on Scientific Computing **1** (1980), 48–53.

[BMR+08] F Buschmann, R Meunier, H Rohnert, P Sommerlad, and M Stal, *Pattern-oriented software architecture: a system of patterns, volume 1*, Wiley India Pvt. Ltd., 2008.

[Boe79] B. Boehm, *Software engineering: R & d trends and defense needs*, Research Directions in Software Technology (Cambridge, Mass.) (MIT Press, ed.), 1979, pp. 44–86.

[BS97] A. Bouaricha and R.B. Schnabel, *Algorithm 768: Tensolve: A software package for solving systems of nonlinear equations and nonlinear least-squares problems using tensor methods*, ACM Transactions on Mathematical Software (TOMS) **23** (1997), no. 2, 174–195.

[CB80] S.D. Conte and C.W.D. Boor, *Elementary numerical analysis: an algorithmic approach*, McGraw-Hill Higher Education, 1980.

[CGT92] A.R. Conn, N.I.M. Gould, and P.L. Toint, *Lancelot: a fortran package for large-scale nonlinear optimization (release a)*, vol. 17, Springer-Verlag Heidelberg, Berlin, New York, 1992.

[CHW02] J. Coplien, D. Hoffman, and D. Weiss, *Commonality and variability in software engineering*, Software, IEEE **15** (2002), no. 6, 37–45.

[CMI05] T. Cickovski, T. Matthey, and J.A. Izaguirre, *Design patterns for generic object-oriented scientific software*, Tech. Report TR05-12, University of Notre Dame, Departament of Computer Science and Engineering, 2005.

[Cop00] J.O. Coplien, *Multi-paradigm design*, Ph.D. thesis, Vrije Universiteit Brussel, 2000.

[DLh+94] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozoz, and K. Remingtonx, *A sparse matrix library in c++ for high performance architectures*, Proceedings of the 2nd AnnualObject-Oriented Numerics Conference, 1994, pp. 214–218.

[DS96] J.E. Dennis and R.B. Schnabel, *Numerical methods for unconstrained optimization and nonlinear equations*, Society for Industrial Mathematics, 1996.

[FK05] W.B. Frakes and K. Kang, *Software reuse research: Status and future*, IEEE Transactions on Software Engineering **31** (2005), no. 7, 529–536.

[Fow97] M. Fowler, *Analysis patterns: reusable object models*, Addison-Wesley, Menlo Park, CA, 1997.

[Fow03] ———, *Patterns [software patterns]*, Software, IEEE **20** (2003), no. 2, 56–57.

[Gh92] J.C. Gilbert and J. Nocedal, *Global convergence properties of conjugate gradient methods for optimization*, SIAM Journal on Optimization **2** (1992), no. 1, 21–42.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-wesley, Massachusetts, 1995.

[GO92] G.H. Golub and J.M. Ortega, *Scientific computing and differential equations: an introduction to numerical methods*, Academic Pr, 1992.

[GOS92] S.I. Grossman, M.G. Osuna, and F.P. Soto, *Álgebra lineal*, McGraw-Hill, 1992.

[GPS99] M.S. Gockenbach, M.J. Petro, and WW Symes, *C++ classes for linking optimization with complex simulations*, ACM Transactions on Mathematical Software (TOMS) **25** (1999), no. 2, 191–212.

[Has76] Lawrence Hasdorff, *Gradient optimization and nonlinear control*, Krieger Pub Co, 1976.

[Kel99] C.T. Kelley, *Iterative methods for optimization*, Society for Industrial Mathematics, 1999.

[Kel03] _____, *Solving nonlinear equations with newton's method*, Society for Industrial Mathematics, Philadelphia, 2003.

[Kir04] D.E. Kirk, *Optimal control theory: an introduction*, Dover Pubns, 2004.

[Mar03] R.C. Martin, *Agile software development: principles, patterns, and practices*, Prentice Hall PTR Upper Saddle River, NJ, USA, 2003.

[MBT04] Kaj Madsen, Hans Bruun, and Ole Tingleff, *Methods for non-linear least squares problems*, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2004.

[MCH+04] T. Matthey, T. Cickovski, S. Hampton, A. Ko, Q. Ma, M. Nyerges, T. Raeder, T. Slabach, and J.A. Izaguirre, *Protomol, an object-oriented framework for prototyping novel algorithms for molecular dynamics*, ACM Transactions on Mathematical Software (TOMS) **30** (2004), no. 3, 237–265.

[MGH80] J.J. Moré, B.S. Garbow, and K.E. Hillstrom, *User guide for minpack-i*, Tech. report, Argonne National Laboratory Report No. ANL-80-74, 1980.

[MMG+00] J.E. Moreira, S.P. Midkiff, M. Gupta, P.V. Artigas, M. Snir, and R.D. Lawrence, *Java programming for high-performance numerical computing*, IBM Systems Journal **39** (2000), no. 1, 21–56.

[MOHW07] J.C. Meza, R.A. Oliva, P.D. Hough, and P.J. Williams, *Opt++: An object-oriented toolkit for nonlinear optimization*, ACM Transactions on Mathematical Software (TOMS) **33** (2007), no. 2, 12–27.

[MS83] B.A. Murtagh and M.A. Saunders, *Minos 5. 0 user's guide*, Tech. report, Stanford Univ., CA (USA). Systems Optimization Lab., 1983.

[Nas00] S.G. Nash, *A survey of truncated-newton methods*, Journal of Computational and Applied Mathematics **124** (2000), no. 1-2, 45–59.

[Pow70] M.J.D. Powell, *A hybrid method for nonlinear equations*, Numerical Methods for Nonlinear Algebraic Equations (P. Rabinowitz, ed.), vol. 4, Gordon and Breach, 1970.

[PSS04] A.D. Padula, S.D. Scott, and W.W. Symes, *The standard vector library: a software framework for coupling complex simulation and optimization*, Tech. Report TR05-12, Rice University, 2004.

[PSS09] ———, *A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms*, ACM Transactions on Mathematical Software (TOMS) **36** (2009), no. 2, 8:1–8:36.

[San09] Julio César Pérez Sansalvador, *Arquitectura flexible para métodos de integración multitasa*, Master's thesis, Instituto Nacional de Astrofísica, Óptica y Electrónica, 2009.

[SFG+06] S.R. Schach, E. Fernández, E. Guerrero, R.A.T. Ramírez, and S.T.J. Betancourt, *Ingeniería de software clásica y orientada a objetos*, McGraw-Hill, 2006.

[Som04] I. Sommerville, *Software engineering*, 8th ed., Addison-Wesley, 2004.

[ST01] A. Shalloway and J. Trott, *Design patterns explained: A new perspective on object-oriented design*, Addison-Wesley, Menlo Park, CA, 2001.

[Vli98] J. Vlissides, *Pattern hatching: design patterns applied*, Addison-Wesley Longman Ltd. Essex, UK, UK, 1998.

[WBM87] L.T. Watson, S.C. Billups, and A.P. Morgan, *Algorithm 652: Hompack: A suite of codes for globally convergent homotopy algorithms*, ACM Transactions on Mathematical Software (TOMS) **13** (1987), no. 3, 281–310.

[WhN99] S. Wright and J. Nocedal, *Numerical optimization*, Springer verlag, 1999.

[XZ01] C. Xu and J. Zhang, *A survey of quasi-newton equations and quasi-newton methods for optimization*, Annals of Operations research **103** (2001), no. 1, 213–234.

Apéndice A

Distribución de las clases en los paquetes

En este apéndice se encuentra la distribución de las clases en los paquetes y una breve descripción de cada clase.

A.1. Paquete *Default*

Main
Clase principal del programa

A.2. *Paquete algebralineal*

Matriz: esta clase es una interfaz para utilizar paqueterías de algebra lineal que implementen operaciones con matrices.

ResSistemaLineal: esta clase es una interfaz para resolver sistemas lineales usando descomposición LU.

SVD: ésta es una interfaz para adaptar paquetes de algebra lineal que realicen la descomposición de valores singulares (SVD).

AdaptadorJaMaLU: esta clase es un Adaptador para resolver sistemas lineales por descomposición LU y usando el paquete de algebra lineal *JaMa: Java Matrix Package*.

AdaptadorJaMaMatrix: esta clase es un adaptador para la clase Matrix del paquete *JAMA: Java Matrix Package*.

AdaptadorSVD: éste es un adaptador para la clase *SingularValueDecomposition* del paquete *JaMa: java matrix package.*

FbAlgebraLineal: esta clase está encargada de instanciar el paquete de algebra lineal usado en la arquitectura para Newtons

A.3. Paquete *arquitecturabase*

ComponentesNewton: esta clase es una fachada para las clases *Funcion, LongitudDePaso, DireccionNewton y CriterioDeParo.*

CriterioDeParo: esta clase es la raíz de las clases que definen los distintos criterios de paro en el método Newton.

DireccionNewton: esta clase es la raíz de las clases que definen los distintas formas de calcular la dirección Newton.

Funcion: esta clase es la raíz para una función $F : \mathbb{R}^n \to \mathbb{R}^m$.

LongitudDePaso: esta clase es la raíz de las clases que definen las distintas formas de calcular la longitud de paso en el método Newton.

Newton: esta clase representa el nivel más alto de abstracción en la arquitectura Newton.

NewtonGenerico: esta clase es la abstracción dentro del patrón Puente.

NewtonImp: esta clase es la implementación dentro del patrón Puente en la arquitectura Newton.

A.4. Paquete *criteriosdeparo*

MiCriterioDeParo: esta clase es un criterio de paro burdo (no hace nada), tan sólo se creo para considerar este aspecto del método Newton dentro de la arquitectura.

A.5. Paquete *derivadas.*

BFGS: esta clase es una implementación de la actualización de la matriz Hessiana en los métodos BFGS.

DerivadaPrimerOrden: esta clase es la raíz para una familia de clases que

implementen algoritmos para calcular gradiente, matriz jacobiana, matrices Quasi-Newton, etc..

DerivadaSegundoOrden: ésta es la clase raíz de una familia de clases para aproximar una derivada de segundo orden, ya sea por una actualización Quasi-Newton, diferencias finitas o analíticamente.

DFuncionConteBoor: derivada analítica de la función del libro de Conte y Boor *"Elementary Numerical Analysis An Algorithmic Approach"* especificada en la clase *FuncionConteBoor*.

Diferencias: ésta es una implementación sencilla para aproximar por diferencias finitas.

DiferenciasGustavo ésta es una implementación para derivar por diferencias finitas.

DMiFuncion: esta clase contiene la derivada de arc tan x.

FbDerivadasPrimerOrden: esta clase se encarga de instanciar las clases en las que se definen los algoritmos usados para calcular derivadas de primer orden.

FbDerivadasSegundoOrden: esta clase se encarga de instanciar las clases en las que se definen los algoritmos usados para calcular derivadas de segundo orden.

HessianaDiferencias: ésta es una implentación simple para calcular una segunda derivada con diferencias finitas.

HessianaIdentidad: ésta es una matriz identidad que utiliza la interfaz de DerivadaSegundoOrden.

QuasiBroyden: ésta es una implementación simple de una fórmula Quasi Newton para aproximar una matriz Jacobiana.

A.6. Paquete *direcciones*

DireccionNewtonRegion: éste es un adaptador de una dirección Newton sujeta al radio de confianza usado en los algoritmos de región de confianza.

DireccionOptimizacion: ésta es una implementación de la dirección Newton : $Hx = -g$, donde H es la hessiana de la función y g es el gradiente. Esta dirección es usada en los métodos Newton para problemas de mínimos cuadrados y optimización sin restricciones.

DireccionRegion: Esta clase sirve como un adaptador para "sujeta" una

dirección newton al radio de confianza usado en los algoritmos de regiones de confianza.

DireccionSistemaNoLineal: ésta es una implementación de la dirección Newton usada en la resolución de sistemas no lineales $Jx = F$, donde J es la jacobiana de la función y F es la función evaluada en la iteración actual.

EmpinadaDescendente: ésta es una implementación de un gradiente descendente usado en los problemas de sistemas no lineales, mínimos cuadrados y optimización sin restricciones.

MLinealAloLargoSpestDescent: esta clase encuentra el minimizador del modelo lineal $M(x + p) = Fx + Jp$, $p = x - x_n$ a lo largo de la dirección del gradiente descendente.

PuntoCauchy: ésta es una implementación del punto Cauchy usado en los algoritmos de región de confianza $p = -a(r/||g||)g$, a es un escalar que minimiza al modelo cuadrático a lo largo del gradiente descendente, r es el radio de confianza g es el gradiente.

A.7. Paquete *fabricaabstracta*

Observador: esta clase forma parte del patrón Modelo Vista Controlador.

FabricaAbstracta: ésta es una fábrica abstracta que sirve como raíz para las diversas fábricas encargadas de instanciar la arquitectura para métodos del tipo Newton

FachadaDeFabricas: esta clase es una fachada para las distintas fábricas encargadas de instanciar la arquitectura para métodos Newton.

Instancia: esta clase se utiliza para asociar a cada posible instancia, usada en la arquitectura, con su fábrica correspondiente.

Modelo: esta clase es la raíz para los distintos modelos para la interfaz gráfica (no confundir con los modelos lineales y cuadráticos) utilizados en la arquitectura Newton.

A.8. Paquete *funciones*

Chebyquad: ésta es una implementación de la funcion Chebyquad usada en el artículo *"Robust Methods for Solving Systems of Nonlinear Equations"* de James

L. Blue.

FbFunciones: esta clase se encarga de instanciar la función a utilizar en la arquitectura para Newtons.

FuncionConteBoor: ésta es la función utilizada para probar un newton Amortiguado.

FuncionPowell : esta función es utilizada para probar el algoritmo de Powell del artículo *"A Hybrid Method for Nonlinear Equations"* y *"A Fortran Subroutine for Solving Systems of Nonlinear Algebraic Equations"* ambos artículos vienen en el libro *"Numerical Methods for Nonlinear Algebraic Equations"*.

FuncionZhanlav: esta función aparece en el paper "*On Newton Type Methods with Four and Fifth Order Convergence"* de Zhanlav.

MiFuncion: ésta es una función para hacer pruebas, su derivada está en la clase DMiFuncion.

MiFuncion3: ésta es una función para hacer pruebas.

MotorDC: ésta clase es un adaptador para evaluar la funcion Costo del Modelo de un MotorDC.

ParaboloideEliptico: ésta función es un paraboloide elíptico que se "ensancha" o "aplasta" con un factor k.

A.9. Paquete *JacobianMatriz.Abstraccion*

Derivada: ésta es la clase raíz para definir derivadas.

Funcion: ésta es la clase raíz para definir funciones de la forma $f : \mathbb{R}^n \to \mathbb{R}$.

FuncionVectorial: ésta es la clase raíz para definir funciones de la forma $f : \mathbb{R}^n \to \mathbb{R}^m$.

MatrizJacobiana: ésta es la clase raíz para definir distintas estrategias para calcular una matriz jacobiana.

A.10. Paquete *JacobianMatriz.principal*

DerivadaConcreta: esta clase sirve para calcular la derivada de una función de una variable.

MiFuncion: esta clase contiene una función para hacer pruebas.

MiFuncionVectorial: esta clase contiene una función para hacer pruebas.

MiMatrizJacobiana: esta clase contiene un algoritmo de diferencias finitas para calcular una matriz jacobiana.

Prueba: esta clase contiene un ejemplo sencillo para utilizar la clase DerivadaConcreta.

Prueba2: esta clase contiene un ejemplo para utilizar la clase MiMatrizJacobiana.

Vector: esta clase define un las operaciones básicas de un vector.

A.11. Paquete *lineasdebusqueda*

Backtracking: esta clase es parte de un algoritmo de línea de búsqueda que utiliza backtracking

CondicionArmijo: ésta es la condición de decremento usada en la regla Armijo ver el libro de Kelley *"Solving nonlinear equations with Newton's Method"* página 11.

CondicionDecremento: esta clase es parte del patrón Estrategia usado para intercambiar distintas condiciones de decremento en líneas de búsqueda.

CondicionSimple: esta clase contiene una condición de decremento simple $F(x_n) < F(x_{n+1})$, es usada para las líneas de búsqueda y es parte del patrón estrategia.

EstrgsCondicionDecremento: esta clase es una interfaz para la familia de clases que definen distintas condiciones de decremento en líneas de búsqueda.

EstrgsLineasBusqueda: esta clase es una interfaz para la familia de clases que definen distintas estrategias para buscar longitudes de paso en líneas de búsqueda.

FbLineasBusqueda: esta clase está encargada de crear los distintos algoritmos de líneas de búsqueda.

LBReglaArmijo: esta clase define una línea de búsqueda que usa la regla armijo.

LineaBurda: ésta es una línea de búsqueda "burda" tan sólo sirve para facilitar la evaluación de métodos con y sin línea de búsqueda.

LineaDeBusqueda: ésta clase es parte del patrón Estrategia, sirve para poder intercambiar de manera sencilla entre distintas líneas de búsqueda.

A.12. Paquete *motor*

BotonRojo: en esta clase se asignan todos los parámetros, el integrador y el método de optimización usado para encontrar los paramétros óptimos en el modelo del motor DC.

FuncionCosto: esta clase es la función costo del mortor DC.

IntegrarModelo: esta clase se encarga de integrar el modelo con el método de integración indicado.

Modelo: en esta clase se evalúan las derivadas que conforman el modelo del motor DC.

Parametros: en esta clase se especifican todos los parámetros que ocupa el modelo.

A.13. Paquete *newtonsconcretos*

FbTiposNewton: esta clase se encarga de instanciar el tipo de Newton a utilizar en la arquitectura para Newtons.

GradienteDescendente: ésta es la implemtación de un gradiente descendente que usa una línea de búsqueda especificada por la fábrica FbLineasBusqueda.

NewtonCauchy: esta clase es un método Newton que utiliza un punto Cauchy y un algoritmo de regiones de confianza.

NewtonClasico: esta clase es la implementación de un Newton clásico que utiliza una dirección para sistemas no lineales y una línea de búsqueda especificada por la clase FbLineasBusqueda.Usa descomposición LU para resolver el sistema lineal en la direccion Newton.

NewtonHibridoBlue: ésta es una implementación del método descrito en el paper "*Robust Newton Methods for Solving Systems of Nonlinear Equations*" de James Blue.

NewtonOptimizacion: este es un método newton que utiliza una dirección x, $Hx = g$, donde H es la hessiana y g es el gradiente, puede utilizar la línea de búsqueda especificada por la clase FbLineasBusqueda.

NewtonPataNocedal: éste es un método "dogleg" que usa actualización de región de confianza como se especifica en el libro *"Numerical Optimization*" de Jorge Nocedal.

PataPowell: este es una implementación del método newton que se describe en los artículos *"A Hybrid Method for Nonlinear Equation"* y *"A Fortran Subroutine for Solving Systems of Nonlinear Algebraic Equations"* ambos artículos vienen en el libro *"Numerical Methods for Nonlinear Algebraic Equations"*.

A.14. Paquete *PrecisionNumerica*

MiPrecisionNumerica: esta clase contiene un método simple para encontrar el épsilon de la máquina.

A.15. Paquete *problemasnolineales*

FbProblemaNoLineal: esta clase se encarga de instanciar el tipo de problema no lineal en la arquitectura para Newtons.

MinimosCuadrados: esta clase define el comportamiento de la clase Funcion de acuerdo al problema de mínimos cuadrados.

MiPortaPapeles: esta clase es un patrón singleton en donde se almacenan los datos más utilizados en el método, por ejemplo el valor de la función en el punto x_n, la primera derivada de la función, el punto de inicio, etc.

OptimizacionSinRestricciones: esta clase define el comportamiento de la clase Funcion, de acuerdo al problema de optimiziación sin restricciones.

PortaPapeles: esta clase es la raiz de las clases encargadas de almacenar datos como el valor de la funcion en cada iteración, el punto de inicio, etc.

ProblemaNoLineal: esta clase es la raíz para los distintos problemas no lineales.

SistemasNoLineales: esta clase define el comportamiento de la clase Funcion, de acuerdo al problema de resolución de sistemas no lineales.

A.16. Paquete *RegionesDeConfianza*

ActualizacionRegionNocedal: en esta clase se encuentra un algoritmo de actualización de región de confianza programado como se especifica en el libro "*Numerical Optimization*" de Jorge Nocedal.

HibridoDeBlue: en esta clase se encuentra un algoritmo que hace el calculo de la dirección newton como se especifica en el artículo "*Robust Methods for Systems of Nonlinear Equations*".

PataDePerroNocedal: esta clase contiene un algoritmo "dogleg" como se especifica en el libro *"Numerical Optimization*" de Jorge Nocedal.

PataDePerroPowell: en esta clase se encuentra un algoritmo tipo "pata de perro" que hace una transición entre el gradiente descendente y un método Newton para sistemas no lineales.

RegionCauchy: esta clase contiene un algoritmo de región de confianza que utiliza una dirección cauchy o punto cauchy.

RegionDeConfianza: esta es la raiz de la familia de métodos que utilizan regiones de confianza.

StrActRadioConfianza: esta clases es parte del patrón estrategia.

A.17. Paquete *vistas*

Controlador: esta clase controla la interacción entre la clase *ModeloNewton*, las fábricas usadas en la arquitectura y la clase Vista usada para presentar los datos.

Grafica: esta clase es la base para implementar gráficas en la arquitectura.

Grafica2: esta clase es un panel que contiene la gráfica que compara número de evaluaciones de la función objetivo y $log_{10}||Fx||$.

Grafica3: esta clase es un panel que contiene la gráfica de el tiempo de ejecución contra $log_{10}||Fx||$.

ModeloNewton: esta clase es el modelo dentro del patrón modelo vista controlador.

PuntoInicio: esta es una pequeña ventana que permite escribir el punto de inicio o seleccionarlo desde un archivo.

Vista: esta clase es una vista de la arquitectura Newton.

Apéndice B

Abstracción e inestabilidad en cada paquete

A continuación se muestran, a detalle, los resultados de medir la abstracción e inestabilidad de la arquitectura. Lo presentado es el resultado de medir la arquitectura mientras ésta crecía. Primero se muestra la arquitectura conformada con sólo cinco paquetes, y por último se presenta el estado de la arquitectura hasta la escritura de esta tesis. Las métricas usadas se explican en la sección 2.2.6 del marco teórico.

- Nc es el número de clases en el paquete.
- Na es el número de clases abstractas en el paquete.
- Ca es el acoplamiento aferente: número de clases fuera del paquete que dependen de las clases dentro del paquete.
- Ce es el acoplamiento eferente: número de clases adentro del paquete que dependen de clases afuera de este paquete.

B.1. Arquitectura base y cuatro paquetes principales

Tabla B.1: Inestabilidad y abstracción de la arquitectura base y cuatro paquetes principales

Paquete	Nc	Na	Ca	Ce	A	I
arquitecturabase	8	6	4	1	0.75	0.2
criteriosparo	1	0	0	1	0	1
direcciones	1	0	0	1	0	1
funciones	1	0	0	1	0	1
lineasdebusqueda	1	0	0	1	0	1

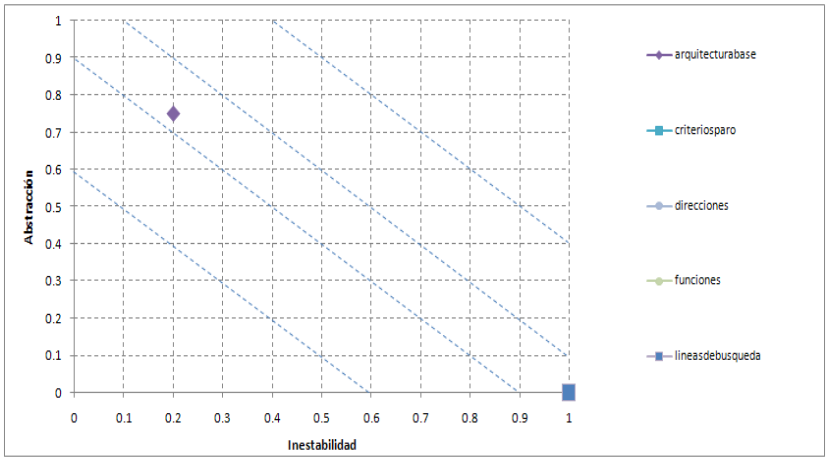


Figura B.1: Inestabilidad y abstracción de la arquitectura base y cuatro paquetes principales.

B.2. Paquetes con el diseño de líneas de búsqueda incorporado

Tabla B.2: Inestabilidad y abstracción de los paquetes tomando en cuenta el diseño de líneas de búsqueda.

Paquete	Nc	Na	Ca	Ce	A	I
Default	1	0	0	4	0	1
PrecisionNumerica	1	0	2	1	0	0.33
algebralineal	5	2	7	4	0.4	0.36
arquitecturabase	8	6	8	5	0.75	0.38
criteriosdeparo	1	0	0	1	0	1
derivadas	4	1	3	7	0.25	0.7
direcciones	1	0	1	4	0	0.8
fabricaabstracta	5	3	10	8	0.6	0.44
funciones	2	0	3	4	0	0.57
lineasdebusqueda	7	2	3	7	0.29	0.7
newtonsconcretos	2	0	3	7	0	0.7
problemasnolineales	7	2	6	6	0.29	0.5
vistas	12	0	1	19	0	0.95

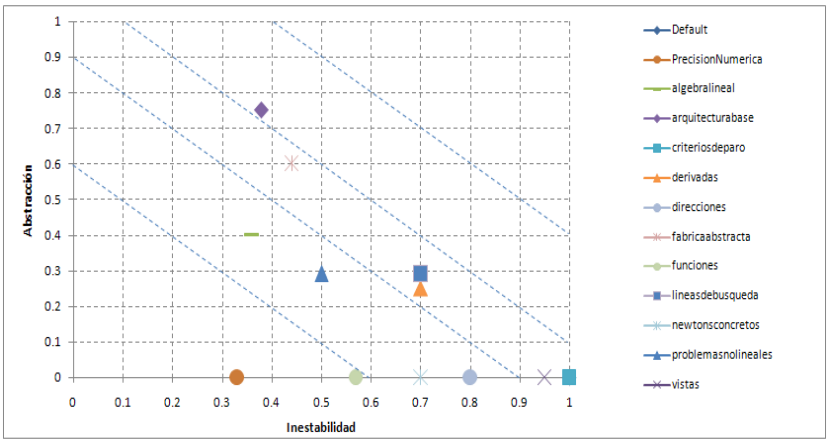


Figura B.2: Inestabilidad y abstracción de los paquetes tomando en cuenta el diseño de las líneas de búsqueda.

B.3. Paquetes con el diseño de líneas de búsqueda y regiones de confianza

Tabla B.3: Inestabilidad y abstracción de los paquetes tomando en cuenta el diseño de líneas de búsqueda y regiones de confianza

Paquete	Nc	Na	Ca	Ce	A	I
Default	1	0	0	4	0	1
PrecisionNumerica	1	0	3	1	0	0.25
RegionesDeConfianza	7	2	2	8	0.29	0.8
algebralineal	7	3	8	4	0.43	0.33
arquitecturabase	8	6	9	5	0.75	0.36
criteriosdeparo	1	0	0	1	0	1
derivadas	6	1	3	7	0.17	0.7
direcciones	7	1	2	6	0.14	0.75
fabricaabstracta	5	3	11	8	0.6	0.42
funciones	8	0	3	4	0	0.57
lineasdebusqueda	8	2	3	7	0.25	0.7
newtonsconcretos	8	0	3	8	0	0.73
problemasnolineales	7	2	7	6	0.29	0.46
vistas	12	0	1	19	0	0.95

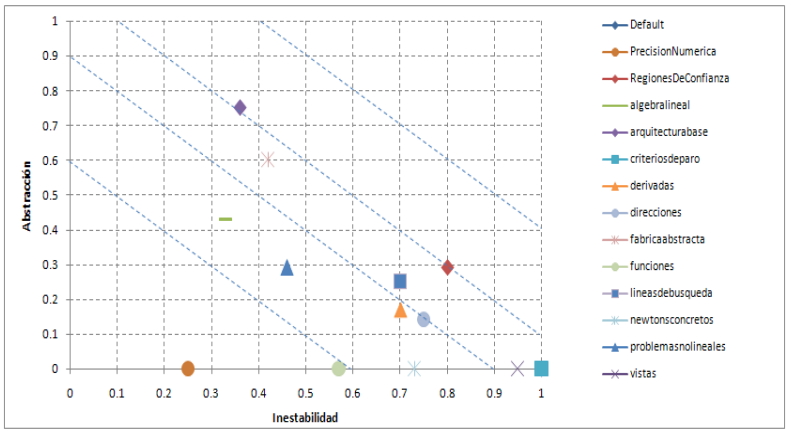


Figura B.3: Inestabilidad y abstracción de los paquetes al agregar el diseño de lineas de búsqueda y regiones de confianza.

B.4. Todos los paquetes que conforman la arquitectura (hasta la escritura de esta tesis)

Tabla B.4: Inestabilidad y abstracción de todos los paquetes que conforman la arquitectura hasta la escritura de esta tesis

Paquete	Nc	Na	Ca	Ce	A	I
Default	1	0	0	4	0	1
JacobianMatriz.Abstraccion	4	4	1	2	1	0.67
JacobianMatriz.principal	7	0	1	5	0	0.83
motor	5	0	1	5	0	0.83
PrecisionNumerica	1	0	3	1	0	0.25
RegionesDeConfianza	7	2	2	8	0.29	0.8
algebralineal	7	3	8	4	0.43	0.33
arquitecturabase	8	6	11	5	0.75	0.31
criteriosdeparo	1	0	0	1	0	1
derivadas	12	2	3	8	0.17	0.73
direcciones	7	1	2	6	0.14	0.75
fabricaabstracta	5	3	11	8	0.6	0.42
funciones	9	0	3	5	0	0.62
lineasdebusqueda	10	2	3	7	0.2	0.7
newtonsconcretos	8	0	3	8	0	0.73
problemasnolineales	7	2	7	6	0.29	0.46
vistas	12	0	1	19	0	0.95

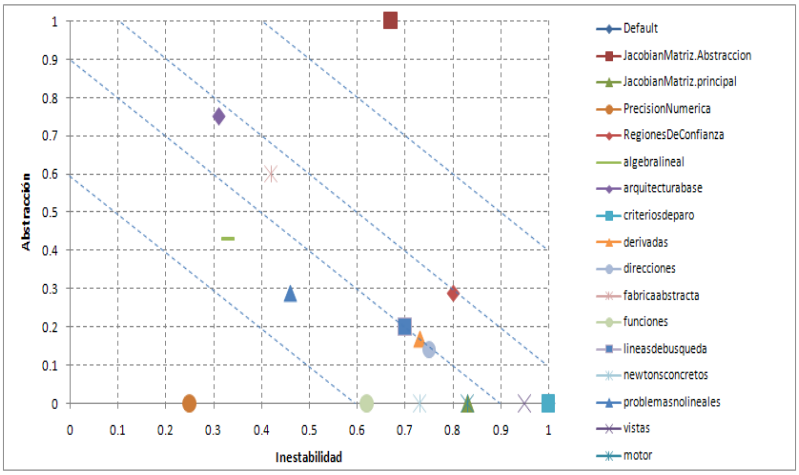
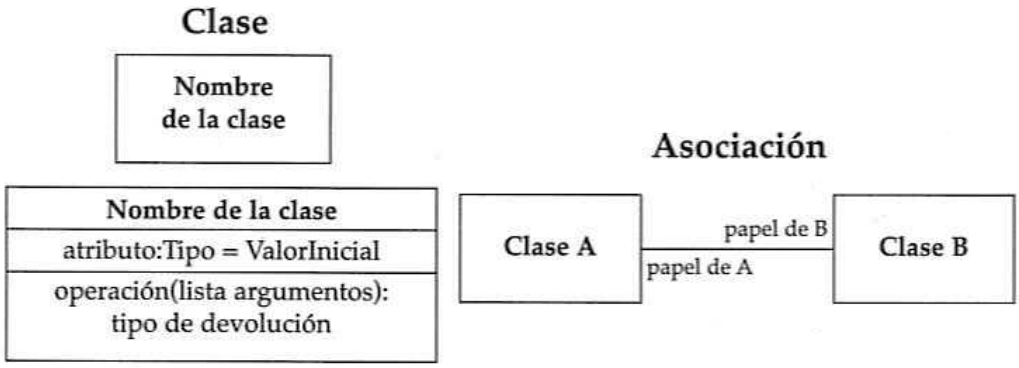
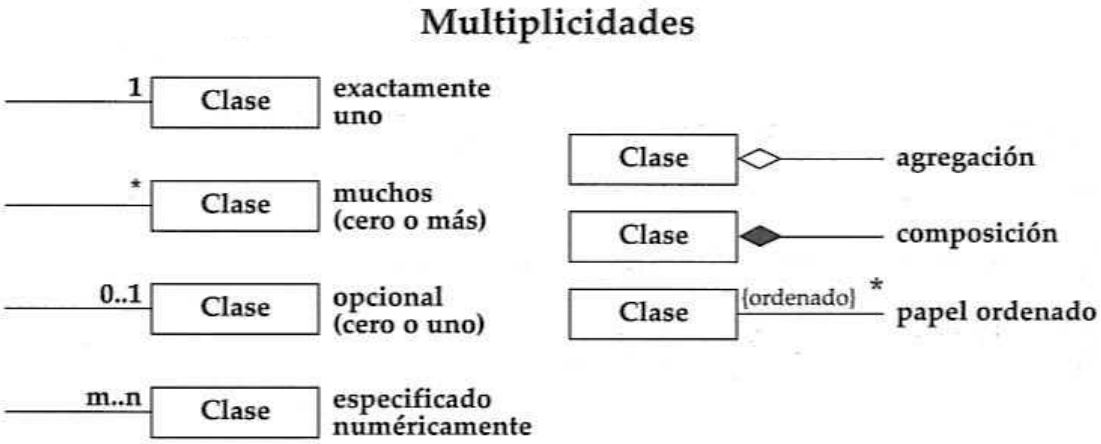


Figura B.4: Inestabilidad y abstracción de todos los paquetes que conforman la arquitectura hasta la escritura de esta tesis.

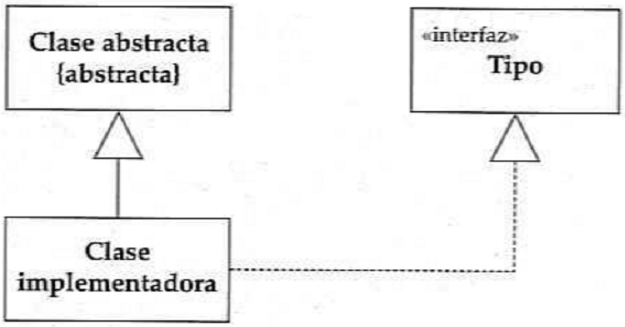
Apéndice C

Resumen de notación UML

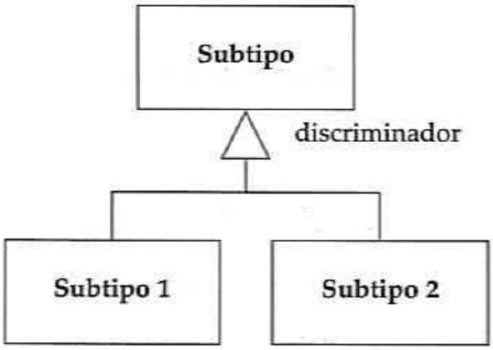




Interfaces

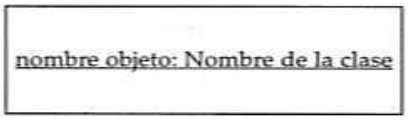


Generalización



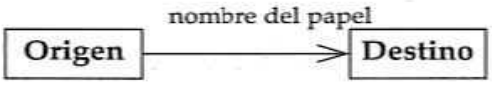
Nota

Objeto



Asociación calificada

Navegabilidad



Dependencia