Agradecimentos

Os meus maiores agradecimentos vão para a Doutora Teresa Monteiro, orientadora deste projecto. A sua ajuda em termos científicos e humanos foram imprescindíveis. Desde o primeiro momento apoiou-me com a sua total disponibilidade e presença, lançando novas ideias, críticas e sugestões, corrigindo e incentivando de forma expedita e eficaz. Além disso, não posso deixar de salientar o carinho e compreensão que sempre me dedicou, possibilitando o desenvolvimento de uma relação de amizade. Sem a sua alegria e motivação não teria sido possível terminar este trabalho. Muito obrigada por tudo!

Não posso deixar de agradecer ao Doutor Ismael Vaz pela preciosa ajuda que me foi dando ao longo do desenvolvimento deste trabalho.

Também quero agradecer aos meus pais e às minhas irmãs, por todo o carinho, compreensão e apoio incondicional que sempre me deram em todas as minhas decisões. A eles dedico este trabalho.

Quando menos esperamos a vida coloca diante de nós um desafio
para testar a nossa coragem e a nossa vontade de mudança.

Paulo Coelho

Resumo

O objectivo deste trabalho é a implementação de um algoritmo para resolução de problemas de optimização não linear com restrições, utilizando o método dos filtros numa abordagem de restauração inexacta (IR, do inglês *Inexact Restoration*).

Na abordagem IR, existem duas fases totalmente independentes em cada iteração - a admissibilidade e a optimalidade. A primeira tem como objectivo, conduzir o processo iterativo na direcção da região admissível, *i.e.*, encontrar um ponto que viole menos as restrições. A partir deste ponto intermédio, efectua-se a fase da optimalidade, consistindo na optimização da função objectivo, no espaço das restrições satisfeitas. Na fase da admissibilidade, resolve-se um problema de optimização linear, cuja função objectivo é o somatório da linearização das restrições violadas, e cujas restrições são a linearização das restrições verificadas. A fase da optimalidade, consiste na resolução de um problema de optimização quadrático, cuja função objectivo é uma aproximação quadrática a partir do ponto obtido na fase anterior e as restrições são a linearização das restrições satisfeitas no mesmo ponto.

Para avaliar as aproximações à solução calculadas em cada iteração, é utilizado um esquema baseado no método dos filtros, nas duas fases do algoritmo. Este método substitui as funções mérito, baseadas em esquemas de penalidade, evitando os inconvenientes associados, tais como a estimação do parâmetro de penalidade e a não diferenciabilidade de algumas delas. Implementou-se o método dos filtros no âmbito da técnica de globalização de procura unidimensional.

Nas experiências computacionais, resolveram-se problemas teste codificados em AMPL. Para inferir a importância da fase de admissibilidade, foram realizados testes com duas versões do algoritmo desenvolvido - uma com as duas fases, a segunda apenas com a fase de optimalidade. Comparou-se o algoritmo com os pacotes de *software* LOQO e NPSOL.

Abstract

The purpose of this work is to develop an algorithm to solve nonlinear constrained optimization problems, using the filter method with inexact restoration (IR) approach.

In the IR approach two independent phases are performed in each iteration - the feasibility and the optimality phases. The first one directs the iterative process into the feasible region, *i.e.*, finds one point with less constraints violation. The optimality phase starts from this point and its goal is to optimize the objective function into the satisfied constraints space. At the feasibility phase an optimization problem is solved, whose objective function is the sum of the violated constraints linearization and the constraints are the satisfied constraints linearization. The optimality phase optimizes a quadratic approximation from the point obtained in the previous phase subject to the set of satisfied constraints in the same point.

To evaluate the solution approximations in each iteration a scheme based on filter method is used in both phases of the algorithm. This method replaces the merit functions that are based on penalty schemes, avoiding the related difficulties such as the penalty parameter estimation and the nondiferentiability of some of them. The filter method is implemented in a context of line search globalization technique.

A set of AMPL test problems is solved. To evaluate the relevance of the feasibility phase, two versions of the algorithm are tested - the first one, with feasibility and optimality phases and the second with optimality phase only. The algorithm developed is compared with LOQO and NPSOL software packages.

Índice

Agradecimentos **i**

Resumo **iii**

Abstract **v**

Índice de Tabelas **ix**

Índice de Figuras **xi**

1 Introdução **1**

1.1 Contextualização e objectivos do trabalho 2

1.2 Organização da dissertação 3

2 Problema a resolver **5**

2.1 Definição do Problema 5

2.2 Condições de Optimalidade 6

2.3 Técnicas de Resolução 8

2.3.1 Funções de penalidade sequencial e exacta 9

3 Método dos Filtros **13**

3.1 Conceitos 14

3.2 Estado da Arte 18

4 Técnicas usadas no algoritmo **23**

4.1 Abordagem IR . . . 24

4.2 Procura Unidimensional com Filtros . . . 26

4.3 Algoritmo base . . . 29

5 Algoritmo IR com Filtros **33**

5.1 Componentes do algoritmo . . . 33

5.1.1 Admissibilidade . . . 33

5.1.2 Optimalidade . . . 35

5.1.3 Actualização do filtro . . . 36

5.1.4 Critério de paragem . . . 36

5.2 Algoritmo final . . . 37

5.2.1 Exemplo ilustrativo . . . 39

5.3 Detalhes da implementação . . . 41

6 Resultados Computacionais **47**

6.1 Ambiente de programação e testes numéricos . . . 47

6.1.1 Avaliação da importância da fase da admissibilidade . . . 53

6.1.2 Comparação com o LOQO e NPSOL . . . 55

7 Conclusões e Trabalho Futuro **63**

Bibliografia **65**

Apêndices **71**

A Código Fonte do Algoritmo **73**

Índice de Tabelas

6.1 Resultados numéricos gerais do Algoritmo IR com Filtros 49

6.2 Análise comparativa do Algoritmo IR com Filtros com e sem fase de admissibilidade 54

6.3 Análise comparativa do Algoritmo IR com Filtros com o LOQO e NPSOL 58

Índice de Figuras

3.1 Filtro NLP com quatro pares de pontos 15

3.2 Ponto candidato na região proibida do filtro 15

3.3 Ponto candidato na região proibida do filtro - dominado por dois pontos 15

3.4 Ponto candidato na região proibida do filtro - o ponto é rejeitado 16

3.5 Ponto candidato na região de aceitação do filtro 16

3.6 Ponto candidato na região de aceitação do filtro - ponto domina dois pontos do filtro 16

3.7 Ponto candidato na região de aceitação do filtro - novo filtro 16

3.8 Envelope criado pelas condições de redução suficiente 17

3.9 Filtro com região proibida, região de aceitação e descréscimo significativo . 18

4.1 Esquema da procura unidimensional com filtros 28

4.2 Técnica IR 30

5.1 Fluxograma do Algoritmo IR com Filtros 37

5.2 Ponto temporário inicial x^k 39

5.3 Admissibilidade - ponto intermédio z^k 39

5.4 Optimalidade - nova aproximação x^{k+1} 39

5.5 Análise da necessidade de inserir o ponto intermédio x^k no filtro 39

5.6 Inserção do ponto intermédio x^k no filtro 40

5.7 Remoção dos pontos dominados 40

5.8 Filtro final 40

5.9 Filtro final e novo ponto temporário 40
5.10 Esquema do Algoritmo IR com Filtros com a interface do AMPL e com a subrotina LSSOL 41

6.1 Inserções no filtro (problemas que convergem) 52
6.2 Elementos no filtro final (problemas que convergem) 52
6.3 Inserções no filtro (problemas que não convergem) 53
6.4 Elementos no filtro final (problemas que não convergem) 53
6.5 Perfis de desempenho com e sem admissibilidade 55
6.6 Perfis de desempenho dos três códigos 61

Capítulo 1

Introdução

Neste capítulo faz-se uma introdução à problemática da optimização não linear com restrições. De seguida é efectuada a contextualização do trabalho desenvolvido e apresentada a forma como a dissertação está organizada.

Os modelos de optimização pretendem representar, em termos matemáticos, o objectivo de resolver um problema da melhor forma. O desejo de resolver problemas atingindo o óptimo é tão comum que os modelos de optimização são aplicáveis em áreas do conhecimento tão distintas como a engenharia, a medicina, a economia ou a física, sendo por isso alvo de estudo de muitos investigadores.

Os problemas de optimização são, normalmente, constituídos por três componentes básicos: a função objectivo, que se pretende minimizar ou maximizar, o conjunto de variáveis ou incógnitas que caracterizam o problema e o conjunto de restrições que definem a região de pontos admissíveis. A solução de um problema de optimização consiste num conjunto de valores admissíveis para variáveis, para o qual a função objectivo assume um valor óptimo.

Em termos matemáticos, a optimização pode ser vista segundo duas vertentes: a maximização e a minimização. Estes dois conceitos estão directamente ligados, pois um problema de maximização pode ser transformado num problema de minimização, e vice-versa.

Não existe nenhum algoritmo universal de optimização, pelo contrário, existem vários algoritmos, cada um vocacionado para um tipo particular de problema - a dificuldade

consiste em identificar o melhor algoritmo para resolver um problema específico.

Um dos problemas mais complexos de optimização é o problema de optimização não linear. As funções envolvidas na formulação matemática deste tipo de problema são não lineares nas variáveis - pelo menos algumas das funções de restrição e/ou a função objectivo é não linear. Os métodos de optimização não linear baseiam-se em processos iterativos, isto é, a partir de um aproximação inicial à solução, é gerada uma sequência de aproximações, pela aplicação repetitiva de regras de um algoritmo, que deverá convergir para uma solução do problema ([3], [37], [38], [22]).

1.1 Contextualização e objectivos do trabalho

Este trabalho insere-se no âmbito da dissertação de Mestrado em Engenharia Industrial, ramo Logística e Distribuição, da Universidade do Minho. O tema "Um algoritmo de filtros em optimização não linear: a admissibilidade independente da optimização" enquadra-se num projecto de investigação sobre algoritmos de optimização não linear com restrições utilizando o método dos filtros.

A principal motivação para o desenvolvimento deste trabalho foi o artigo de Gonzaga *et al.* [23] em que é apresentado um algoritmo para resolução de problemas de optimização com restrições baseado no método dos filtros segundo uma abordagem de restauração inexacta (IR, do inglês *Inexact Restoration*).

A técnica dos filtros, introduzida por Fletcher e Leyffer [13], incorpora o conceito de dominância da optimização multi-objectivo, funcionando como esquema de indução de convergência dos algoritmos. Este método surgiu com o objectivo de colmatar os inconvenientes associados às funções de penalidade, tais como a estimação do parâmetro de penalidade e ainda a não diferenciabilidade de algumas delas.

A restauração inexacta foi introduzida por Martínez [34] e trata o problema da admissibilidade e da optimalidade em duas fases distintas. Segundo esta técnica, a admissibilidade é um factor importante do problema e por isso deve ser controlada independentemente da optimalidade.

Os autores em [23], apresentam um algoritmo em alto nível para o qual fazem um estudo teórico de convergência, em que é provada a convergência para um ponto estacionário. No algoritmo proposto, ainda por implementar, não são especificados os métodos internos inerentes a cada uma das fases de admissibilidade e optimalidade - este facto por si só suscitou interesse e funcionou como grande desafio para este trabalho. A implementação numérica do algoritmo, a definição dos procedimentos internos a cada uma da fases e ainda a possibilidade de utilizar a filosofia do método dos filtros, são os principais objectivos deste projecto.

1.2 Organização da dissertação

Este documento está organizado em sete capítulos e um apêndice.

No Capítulo 2 é definido o problema a resolver, o problema de optimização não linear com restrições, apresentando-se alguns conceitos de optimização e ainda as condições de optimalidade usuais. Enumeram-se alguns dos métodos mais utilizados na resolução deste tipo de problemas, dando-se algum relevo à técnica de penalidade. Os inconvenientes desta técnica levaram ao aparecimento de uma nova metodologia apresentada por Fletcher e Leyffer em [13], o método dos filtros, apresentado e explorado com mais detalhe no Capítulo 3.

No Capítulo 4 é apresentada a abordagem de restauração inexacta (IR), a técnica de procura unidimensional no contexto do método dos filtros, o algoritmo base que serviu para a implementação do algoritmo final e ainda alguns tópicos relativos à sua convergência.

O algoritmo implementado, Algoritmo IR com Filtros, é apresentado em detalhe no Capítulo 5.

No Capítulo 6 são apresentadas as experiências computacionais que validam e sustentam o algoritmo desenvolvido e por fim, são apresentadas no Capítulo 7, as conclusões gerais da dissertação e desenhadas as perspectivas de trabalho futuro.

O apêndice mostra o código fonte do algoritmo implementado.

Capítulo 2

Problema a resolver

Neste capítulo apresenta-se a definição do problema a resolver - problema de optimização não linear com restrições. São ainda referidos alguns conceitos genéricos de optimização e enunciadas as condições de optimalidade de primeira e segunda ordem. É feita uma breve explicação da filosofia inerente à técnica de penalidade.

2.1 Definição do Problema

A optimização pode ser vista em duas vertentes - a maximização e a minimização. Os problemas de maximização podem ser postos em termos de problemas de minimização, pelo que o problema de optimização vai ser, a partir de agora, referenciado como de minimização.

Num problema de minimização, o objectivo é encontrar os valores das variáveis que minimizam a função objectivo, satisfazendo as restrições impostas. A formulação geral do problema não linear com restrições (NLP, do inglês *NonLinear Programming*) é a seguinte:

$$\begin{aligned} &\min_{x \in I\!R^n} f(x) \\ \text{s.a} \quad & lb_x \leq x \leq ub_x \\ & lb_c \leq c(x) \leq ub_c \end{aligned} \tag{NLP}$$

onde $f(x) : \mathbb{R}^n \to \mathbb{R}$ denota a função objectivo não linear, lb_x, $ub_x \in \mathbb{R}^n$ são os limites inferior e superior da variável $x \in \mathbb{R}^n$, respectivamente (restrições de limites simples), e $c(x) : \mathbb{R}^n \to \mathbb{R}^m$ representa o conjunto de m restrições não lineares cujos limites inferior e superior são lb_c e ub_c respectivamente. Note-se que as restrições de igualdade também estão incluídas na formulação (NLP), basta que os limites inferior e superior sejam definidos com o mesmo valor para determinada restrição. Do mesmo modo, podem-se considerar restrições limitadas apenas inferior ou superiormente, definindo o limite inferior como $-\infty$ ou superior como $+\infty$. Neste trabalho, as restrições $lb_c \leq c(x) \leq ub_c$ em (NLP) são reformuladas para restrições do tipo $c(x) \leq 0$.

2.2 Condições de Optimalidade

Para enunciar as condições de optimalidade torna-se pertinente a apresentação prévia de algumas entidades e conceitos. Nestes conceitos, por uma questão de simplificação, faz-se referência às restrições como igualdade e desigualdade. Começa então por definir-se a função Lagrangeana.

Função Lagrangeana:

A função Lagrangeana do problema (NLP) é definida por:

$$\mathcal{L}(x, \lambda) = f(x) - \sum_{i \in E \cup I} \lambda_i c_i(x)$$

em que λ é o vector dos multiplicadores de Lagrange, E e I são os conjuntos dos índices das restrições de igualdade e desigualdade, respectivamente.

Em seguida apresentam-se as definições de região admissível e conjunto de restrições activas.

Região admissível:

A região admissível Ω é o conjunto de pontos x que satisfaz todas as restrições, *i.e.*,

$$\Omega = \{x \,|c_i(x) = 0, i \in E;\; c_i(x) \leq 0, i \in I\}$$

Pode então reescrever-se o problema (NLP) como

$$\min_{x\in\Omega} \quad f(x).$$

Conjunto de restrições activas:

Sejam E e I os conjuntos de índices das restrições de igualdade e de desigualdade, respectivamente. Designa-se por conjunto das restrições activas o conjunto:

$$A(x) = \{c_i(x)|i \in E\} \cup \{c_i(x)|c_i(x) = 0 \wedge i \in I\}$$

É usual estabelecerem-se hipóteses, designadas por qualificações das restrições (*constraints qualification*), para garantir que não ocorrerá nenhum comportamento degenerado para o valor de x^*, sendo este a solução do problema em questão. Uma delas, talvez a mais usada na concepção de algoritmos, é a LICQ (*Linear Independence Constraint Qualification*).

LICQ:

Dado um ponto x^* e um conjunto activo $A(x^*)$, a qualificação das restrições de independência linear é satisfeita se o conjunto dos gradientes das restrições activas

$$\{\nabla c_i(x^*), i \in A(x^*)\}$$

é linearmente independente.

Um ponto x que verifica a LICQ diz-se ponto regular, *i.e.*, a matriz $\{\nabla c_i(x), i \in A(x)\}$ tem característica completa (*full rank*). Note-se que se esta qualificação das restrições é satisfeita, nenhum dos gradientes das restrições activas pode ser zero.

Em seguida apresentam-se as condições de optimalidade para o problema (NLP). São designadas de primeira ordem porque dizem respeito às propriedades dos gradientes (primeiras derivadas) da função objectivo e das funções das restrições. Estas condições são conhecidas pelas condições KKT (*Karush Kuhn Tucker*).

Condições de primeira ordem:

Suponhamos que x^* é uma solução local de um problema de optimização não linear e é ponto regular. Então existe um vector dos multiplicadores de Lagrange λ^*, com compo-

nentes λ_i^*, $i \in E \cup I$, tal que as seguintes condições são satisfeitas em (x^*, λ^*):

$$\begin{aligned}
&\nabla_x \mathcal{L}(x^*, \lambda^*) = \nabla f(x^*) - \sum_{i \in E \cup I} \nabla c_i(x^*) \lambda_i^* = 0 \\
&c_i(x^*) = 0,\ i \in E \\
&c_i(x^*) \leq 0,\ i \in I \\
&\lambda_i^* \leq 0,\ i \in I \\
&\lambda_i^* c_i(x^*) = 0,\ i \in E \cup I
\end{aligned} \tag{2.1}$$

O ponto x^* que verifica as condições de primeira ordem diz-se ponto estacionário da Lagrangeana e o par (x^*, λ^*) que as verifica denomina-se par KKT (*Karush Kuhn Tucker*).

As condições seguintes, de segunda ordem, examinam as segundas derivadas e dizem respeito à curvatura da função Lagrangeana.

Condições suficientes de segunda ordem:

O ponto x^*, do par KKT (x^*, λ^*), é um mínimo local se a matriz Hessiana da função Lagrangena $\nabla^2_{xx} \mathcal{L}(x^*, \lambda^*)$ for definida positiva no espaço nulo da matriz $\nabla c(x^*)^T$.

2.3 Técnicas de Resolução

Uma estratégia possível para a resolução do problema (NLP) consiste em transformá-lo num problema sem restrições, de tal forma que a solução deste problema é igual ou está relacionada com a solução de (NLP). Este objectivo pode ser atingido usando várias técnicas, entre as quais:

- métodos baseados em funções de penalidade;
- métodos baseados em funções de barreira;
- método dos gradientes reduzidos;
- método de projecção do gradiente;
- métodos baseados em funções Lagrangeanas aumentadas;

- métodos de Lagrangeanas projectadas;
- método dos filtros.

Estes métodos podem ser aplicados tanto a problemas com restrições do tipo igualdade como desigualdade, havendo para estes últimos várias estratégias de tratamento.

Dar-se-á alguma ênfase às funções de penalidade, uma vez que elas constituem o principal motivo para o aparecimento do método dos filtros. Por uma questão de facilitar a compreensão da filosofia da técnica de penalidade, vão ser apresentadas duas funções de penalidade no âmbito dos problemas de optimização com apenas restrições de igualdade.

2.3.1 Funções de penalidade sequencial e exacta

Uma das formas de resolver um problema de optimização com restrições baseia-se na construção de uma função cujo mínimo é x^*, ou está de alguma forma relacionado com este. O problema original pode ser resolvido pela formulação de uma sequência de problemas sem restrições, cuja solução converge para x^* se a função incluir um termo que garanta a admissibilidade no limite. Uma escolha possível para esse termo define uma função que adiciona à função objectivo uma penalidade positiva sempre que houver violação das restrições.

Considere-se a função de penalidade quadrática representada por

$$P_q(x,r) = f(x) + \frac{1}{2} r c(x)^T c(x) \tag{2.2}$$

em que r é o parâmetro de penalidade ($r > 0$) e o termo de penalidade $\frac{1}{2} r c(x)^T c(x)$ é continuamente diferenciável.

Pode consultar-se em [3] a demonstração de que o mínimo de $P_q(x,r)$ designado por $x^*(r)$ verifica

$$\lim_{r \to \infty} x^*(r) = x^*,$$

ou seja, tende para o mínimo de (NLP) à medida que r aumenta.

Esta técnica, conhecida por técnica de penalidade sequencial, tem o inconveniente de exigir a resolução do problema

$$\min_{x \in I\!R^n} P_q(x,r)$$

para cada valor de r de uma sequência infinita. A sequência $\{x^*(r)\}$ de soluções dos problemas converge para x^*, ou seja, $x^*(r)$ nunca coincide com x^* para valores finitos de r. Além disso, o facto de $r \rightarrow \infty$ pode dar origem a singularidades na matriz Hessiana de $P_q(x, r)$ no limite, tornando o problema mal condicionado e a convergência mais lenta, ou mesmo impossibilitando a convergência para o mínimo pretendido.

O algoritmo de resolução de (NLP) pela utilização desta técnica de penalidade sequencial é:

Algoritmo 2.3.1 *Técnica da Penalidade Sequencial*

x^0, r^0, $k \leftarrow 0$;

Enquanto (x^k *não verifica as condições de optimalidade*) *Fazer*

$\{$*resolução de* $\min\limits_{x \in I\!R^n} P_q(x, r^k)$: $x^*(r^k)\}$

$\{x^{k+1} \leftarrow x^*(r^k)\}$

$\{r^{k+1} \leftarrow r^k + \textit{incremento}\}$

$\{k \leftarrow k + 1\}$

Outra função de penalidade, conhecida por função de valor absoluto é representada por

$$P_A(x, r) = f(x) + r \sum_{i=1}^{m} \mid c_i(x) \mid \tag{2.3}$$

em que r é o parâmetro de penalidade.

Esta formulação tem um inconveniente que reside no facto das derivadas do termo de penalidade serem descontínuas quando $c(x) = 0$.

No problema de minimização sem restrições baseado na função $P_A(x, r)$, a solução $x^*(r)$ coincide com x^* para valores finitos de r. Prova-se em [3] que

$$\underset{r > \bar{r}}{x^*(r)} = x^*.$$

O mau condicionamento verificado no problema com funções de penalidade sequencial é evitado, uma vez que não é necessário exigir que $r \rightarrow \infty$. No entanto, outras dificuldades podem surgir nas funções do tipo $P_A(x, r)$, pois o valor $\bar{r}$ depende de quantidades calculadas

em x^*, que sendo desconhecido, obriga a que r tenha de ser estimado e eventualmente ajustado. Podem surgir outros problemas na implementação se forem escolhidos valores inadequados para r - se r é demasiado pequeno a função de penalização pode ser ilimitada inferiormente ou a região de convergência para x^* ser muito pequena. Por outro lado se r for elevado o problema pode tornar-se mal condicionado.

A principal diferença entre $P_q(x, \rho)$ e $P_A(x, \rho)$ reside no facto de com $P_q(x, \rho)$ o problema original a resolver, problema não linear com restrições de igualdade, ser transformado numa sequência **infinita** de problemas sem restrições, enquanto que com $P_A(x, \rho)$ essa sequência é **finita**. Por esta razão a técnica é designada por penalidade exacta, embora a selecção de um valor adequado para r possa obrigar à resolução de mais do que um problema sem restrições.

O algoritmo para a técnica de penalidade exacta é semelhante ao anterior:

Algoritmo 2.3.2 *Técnica da Penalidade Exacta*

x^0, $\overline{r}$, $k \leftarrow 0$;

Enquanto (x^k *não verifica as condições de optimalidade*) *Fazer*

  {*resolução de* $\min_{x \in I\!R^n} P_A(x, \overline{r})$: $x^*(\overline{r})$}

  {$x^{k+1} \leftarrow x^*(\overline{r})$}

  {*ajuste, se necessário, de* $\overline{r}$}

  {$k \leftarrow k + 1$}

Para os problemas de optimização do tipo (NLP) a função de penalidade apresenta dois termos de penalidade - um para as restrições de igualdade e outro para as de desigualdade. A comunidade científica tem desenvolvido muitos esforços nesta área, tendo surgido muitos trabalho relativos a funções de penalidade. Não é feita uma descrição detalhada sobre este assunto, uma vez que esta técnica não foi utilizada neste trabalho. No entanto, a descrição da filosofia da técnica de penalidade torna-se pertinente uma vez que, a não diferenciabilidade de algumas funções de penalidade e a dificuldade na estimação do parâmetro de penalidade, constituem a principal motivação para o desenvolvimento do método dos filtros, apresentado no capítulo seguinte.

Capítulo 3

Método dos Filtros

Neste capítulo é apresentada, numa primeira fase, a metodologia dos filtros como uma técnica que utiliza o conceito de dominância da optimização multi-objectivo. São introduzidas algumas definições e conceitos inerentes à metodologia, importantes para a compreensão da filosofia do método dos filtros. É ainda efectuada uma revisão por todos os desenvolvimentos e aplicações do método dos filtros por diferentes investigadores em áreas distintas.

O problema da optimização não linear a resolver, definido em (NLP), traz consigo dois objectivos conflituosos para resolver: a minimização da função objectivo e, ao mesmo tempo, a satisfação de todas as restrições. Nas abordagens clássicas, nomeadamente da função penalidade, estes dois objectivos são combinados e transformados num único problema de minimização, utilizando a função mérito.

A estratégia dos filtros utiliza o conceito de dominância da optimização multi-objectivo [36], considerando os dois objectivos separadamente - a minimização da função objectivo $f(x)$ e, ao mesmo tempo, a minimização da violação das restrições. O problema inicial (NLP) de minimização com restrições é transformado em dois problemas de minimização sem restrições, sendo redefinido da seguinte forma:

$$\min_{x \in I\!R^n} f(x) \qquad \text{e} \qquad \min_{x \in I\!R^n} h(c(x)) \tag{3.1}$$

em que $h(c(x)) := \|c^+(x)\|_1 := \sum_{j=1}^{m} c_j^+(x)\ j = 1, \ldots, m$, com $c_j^+(x) = \max(0, c_j(x))$ é a função que representa a soma da violação das restrições. Como já foi referido no Capítulo 2, as restrições $lb_c \leq c(x) \leq ub_c$ em (NLP) são reformuladas para restrições do tipo $c(x) \leq 0$.

3.1 Conceitos

Num filtro são considerados os valores dos pares $(f(x), h(c(x)))$ obtidos pela avaliação de f e h nos vários valores de x. Seguindo Fletcher e Leyffer [13], apresentam-se, de seguida, duas definições importantes para a concepção de um filtro. A primeira diz respeito à estrutura do filtro e a segunda ao conceito de dominância da optimização multi-objectivo.

Definição 3.1.1 *Um par $(f(x^k), h(x^k))$, obtido numa iteração k, diz-se que domina outro par $(f(x^l), h(x^l))$ se e só se $h(x^k) \leq h(x^l)$ e $f(x^k) \leq f(x^l)$.*

A definição acima significa que x^k é pelo menos tão bom como x^l relativamente a ambas as medidas, função objectivo e violação das restrições. Com este conceito pode-se definir o filtro como um critério para aceitar ou rejeitar um passo.

Definição 3.1.2 *Um filtro é constituído por uma lista de pares $(f(x), h(c(x)))$ de tal forma que nenhum par seja dominado por outro. Um par $(f(x), h(c(x)))$ é aceite para inclusão no filtro se este não for dominado por nenhum outro par do filtro.*

Considere-se $\mathcal{F}^k$ a notação que representa um conjunto de iterações de índice j $(j \leq k)$ tal que $h(c(x^j), f(x^j))$ é uma entrada do filtro actual.
Da definição anterior retira-se que um ponto x diz-se "aceite pelo filtro" se se verificar uma das condições

$$h(c(x)) < h(c(x^j)) \quad \textbf{ou} \quad f(x) < f(x^j), \qquad \forall j \in \mathcal{F}^k$$

De salientar que cada vez que um par $(f(x), h(c(x)))$ é inserido no filtro, os pares do filtro que são dominados por este são removidos - o filtro é uma estrutura dinâmica. Deste modo, é garantida a consistência do filtro, podendo ser usado como critério na aceitação

de um passo de um algoritmo.

A figura seguinte representa graficamente um filtro num plano (f, h). Cada ponto do filtro

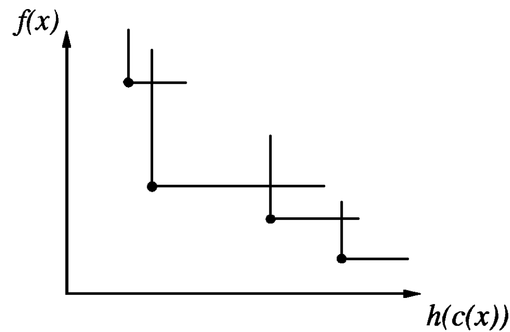

Figura 3.1: Filtro NLP com quatro pares de pontos

gera um bloco de pontos inadmissíveis e a união desses blocos representa a região de pontos inadmissíveis do filtro. De seguida apresenta-se um exemplo ilustrativo do funcionamento geral do filtro.

Considere-se a Figura 3.2 onde estão representados alguns pares num filtro, definindo a região de aceitação e a região proibida. Considere-se também o ponto a azul como um ponto candidato a ser introduzido no filtro. Como se verifica este ponto encontra-se na zona proibida do filtro, isto é, é dominado pelos dois pontos representados a vermelho na Figura 3.3. Deste modo, o par (f, h) correspondente não é aceite pelo filtro e o ponto é rejeitado. O filtro permanece inalterado (Figura 3.4).

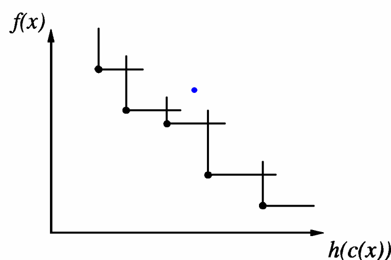

Figura 3.2: Ponto candidato na região proibida do filtro

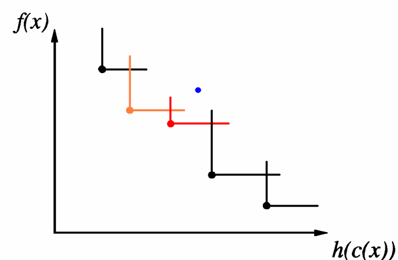

Figura 3.3: Ponto candidato na região proibida do filtro - dominado por dois pontos

Analise-se outro ponto na Figura 3.5. Este ponto encontra-se na região de aceitação e é um ponto melhor do que os dois representados a cinza (Figura 3.6), isto é, o novo par domina dois pares do filtro, logo o ponto é aceite e o correspondente par introduzido no filtro. Ao inserir o novo par no filtro, os pares dominados são removidos. O filtro final fica com uma nova configuração, como representado na Figura 3.7.

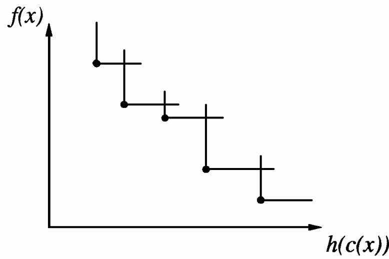

Figura 3.4: Ponto candidato na região proibida do filtro - o ponto é rejeitado

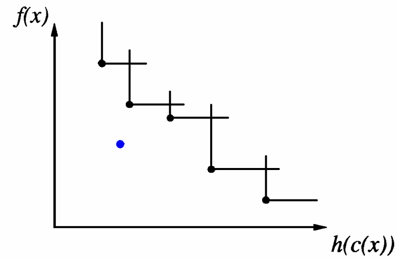

Figura 3.5: Ponto candidato na região de aceitação do filtro

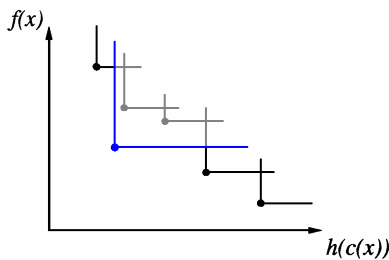

Figura 3.6: Ponto candidato na região de aceitação do filtro - ponto domina dois pontos do filtro

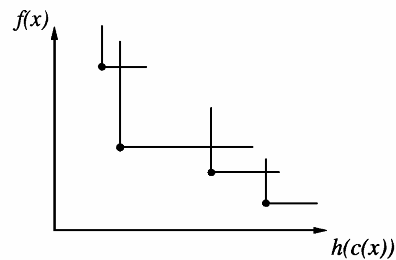

Figura 3.7: Ponto candidato na região de aceitação do filtro - novo filtro

Associado ao filtro está o conceito de *envelope*. Este conceito foi introduzido com o propósito de provar a convergência do método, uma vez que a Definição 3.1 é inadequada pois permite a acumulação de pontos não estacionários, na vizinhança da entrada do filtro. Este facto é facilmente corrigido com a definição de um *envelope* à volta dos pares que constituem o filtro actual, como está demonstrado em [13]. Este conceito está ilustrado na Figura 3.8, onde o *envelope* está representado pela linha a tracejado.

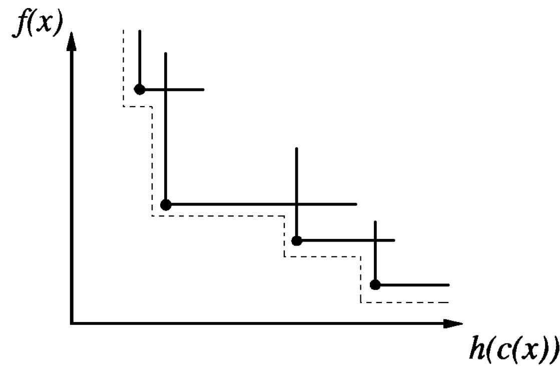

Figura 3.8: Envelope criado pelas condições de redução suficiente

Deste modo, um novo ponto x^{k+1} é aceite pelo filtro se e só se

$$h < \beta h^l \tag{3.2}$$

ou

$$f < f^l - \gamma h \tag{3.3}$$

para todos os pares (f^l, h^l) do filtro, onde β e γ são parâmetros tais que $0 < \gamma < \beta < 1$, com β perto de um e γ perto de zero. A equação (3.2) cria um *envelope* na direcção h do filtro, e a equação (3.3) cria um *envelope* na direcção f do filtro.

O conceito do *envelope* foi introduzido para forçar um decréscimo significativo em h ou f aquando da inserção de pares no filtro. Isto significa que não se pretende que um par seja apenas melhor do que qualquer outro par do filtro, mas que f ou h do novo par sejam suficientemente menores do que uma determinada quantidade.

A ideia chave é utilizar o filtro como um critério para aceitar ou rejeitar um passo - funciona como um oráculo que define uma região *tabu*, como se mostra a vermelho na Figura 3.9. Um novo passo é aceite se reduzir significativamente a função objectivo ou a violação das restrições, o que significa não pertencer à região *tabu*. Voltar-se-á a falar deste conceito no próximo capítulo, no âmbito da técnica de procura unidimensional.

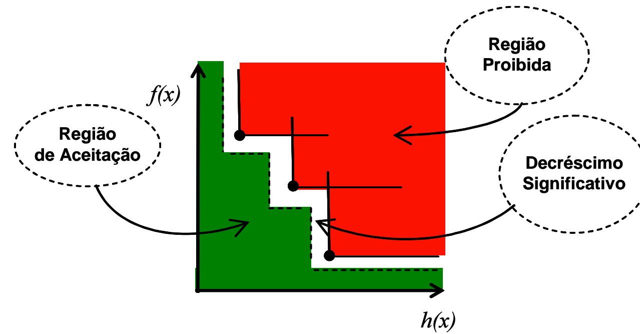

Figura 3.9: Filtro com região proibida, região de aceitação e descréscimo significativo

3.2 Estado da Arte

Esta nova estratégia de globalização dos algoritmos tem suscitado inúmeros trabalhos no âmbito de vários tipos de algoritmos de optimização e nas mais diversas áreas de aplicação.

O método dos filtros foi introduzido por Fletcher e Leyffer [13], para resolver o problema (NLP), como uma alternativa à abordagem tradicional das funções mérito. Os autores utilizam esta filosofia no âmbito de um algoritmo do tipo SQP (do inglês, *Sequential Quadratic Programming*) com a técnica de regiões de confiança. O conceito subjacente é razoavelmente simples - gera uma sequência de novas aproximações à solução do problema a partir da resolução de uma sequência de subproblemas da programação quadrática (QP) com regiões de confiança, sendo os novos pontos aceites pelo algoritmo se produzirem um decréscimo significativo da função objectivo ou da função da violação das restrições. Os resultados computacionais e as comparações com outros *solvers* apresentados em [13] são bastantes encorajadores mostrando que a estratégia dos filtros é competitiva. Consequentemente, a estratégia despertou interesse em vários autores sendo rapidamente apresentada a convergência global de métodos SQP com regiões de confiança em Fletcher *et al.* [15].

Gould e Toint [27] provam a convergência global de uma variante do algoritmo filtro-SQP apresentado por Fletcher *et al.* [15]. Neste algoritmo foram utilizadas duas estratégias no cálculo do passo: a primeira decompõe o passo nas suas componentes normal e tangencial, e a segunda substitui esta decomposição por uma condição mais forte associada com o modelo de decréscimo utilizado.

Ulbrich e Ulbrich [40] analisam uma classe de métodos que utilizam regiões de confiança para resolver problemas de optimização não linear com restrições de igualdade. Apresentam um algoritmo no qual garantem a convergência global sendo ainda apresentados resultados numéricos confirmando a sua robustez e eficiência.

Fletcher *et al.* apresentaram em [14] desenvolvimentos na convergência global de algoritmos de filtros SLP (do inglês, *Sequential Linear Programming*) com regiões de confiança. Também mostram que esta técnica permite um leque de opções de algoritmos específicos na actualização do raio da região de confiança e na restauração da admissibilidade. Chin e Fletcher [9] provam convergência global para uma classe de algoritmos baseados no método dos filtros com regiões de confiança. Adaptaram o algoritmo original filtro-SQP para permitir passos calculados com base no modelo de programação quadrática com restrições de igualdade (EQP, do inglês *Equality Quadratic Programming*).

Em [6] e [7] é avaliado o desempenho de algoritmos de filtros SQP e SLP. Chin e Fletcher [6] apresentam a implementação do algoritmo de filtro SLP com regiões de confiança, com passos EQP, e os resultados computacionais obtidos. O algoritmo SLP utiliza o CPLEX para resolver os subproblemas que possibilitam a obtenção dos passos EQP. Para avaliar o algoritmo são testados problemas da biblioteca CUTE e feitas comparações com os *solvers* filtro-SQP e LANCELOT. O documento [7] funciona como um complemento ao anterior, apresentando os resultados numéricos do algoritmo SLPSQP efectuados com problemas seleccionados da biblioteca CUTE e a sua comparação com os *solvers* filtro-SQP e LANCELOT. Os resultados numéricos que estes apresentam mostram que a estratégia dos filtros tem potencial para competir com as abordagens tradicionais.

Existem outras abordagens utilizando a estratégia dos filtros tais como aplicações aos algoritmos de pontos interiores apresentada por Ulbrich *et al.* [41] e por Benson *et al.* [4]. Ulbrich *et al.* em [41], utilizam a técnica dos filtros num algoritmo primal-dual de pontos interiores. O novo algoritmo decompõe o passo primal-dual obtido das condições de primeira ordem perturbadas em dois passos - o tangencial e o normal, cujos tamanhos são controlados pelo raio da região de confiança. Cada entrada no filtro é um par de coordenadas: uma resultante da admissibilidade e centralidade, e associada com o passo

normal; a outra resultante da optimalidade e relacionada com o passo tangencial.

Em [4], Benson *et al.* analisam várias possibilidades de implementação de uma abordagem baseada nos filtros num algoritmo de pontos interiores. Também efectuam testes numéricos que mostram que a abordagem dos filtros combinada com pontos interiores é mais eficiente do que a utilização da função mérito.

O método *bundle* para optimização não-suave foi inicialmente desenvolvido por Fletcher e Leyffer [12] e explorado por Karas *et al.* em [30]. Neste documento é proposto um algoritmo para resolver problemas de optimização com restrições convexas não-suaves que combina as ideias dos métodos *bundle* proximal com a estratégia dos filtros para avaliar os pontos candidatos. O algoritmo resultante combina propriedades atractivas de ambos os métodos.

Uma abordagem de procura por padrão utilizando o método dos filtros na programação não linear sem derivadas é apresentada por Audet e Dennis [2]. Este documento formula e analisa um método de procura por padrão para a optimização com restrições baseado no método dos filtros para a aceitação do passo. Este algoritmo identifica pontos limite nos quais as condições de optimalidade dependem de funções locais suavizadas. O algoritmo é ilustrado com exemplos de teste e é feita a sua aplicação prática num projecto de desenho em engenharia aeronáutica.

Gould *et al.* em [24] desenvolveram um estudo teórico da convergência global de um algoritmo de filtros com regiões de confiança para optimização sem restrições. Neste trabalho também apresentam alguns resultados numéricos que mostram as potencialidades do método face aos algoritmos clássicos.

A técnica dos filtros também está a ser estudada em algoritmos de procura unidimensional. O primeiro estudo teórico foi apresentado por Gonzaga *et al.* [23]. Neste documento é efectuada uma análise teórica das propriedades de convergência global de um algoritmo de restauração inexacta com procura unidimensional, para problemas não lineares com restrições.

Wätcher e Biegler, seguindo os bons resultados demonstrados por este método, desenvolveram em [44] e [45] um estudo da convergência local e global de um algoritmo com

procura unidimensional, aplicado a métodos barreira com pontos interiores e métodos SQP, utilizando o método dos filtros em substituição da tradicional função mérito. É mostrado que, sob determinadas condições, cada ponto limite gerado pelo algoritmo é admissível, e que pelo menos um desses pontos é estacionário para o problema. Mostra-se também que os métodos propostos não sofrem do efeito Maratos se as direcções de procura forem melhoradas por condições de segunda ordem (SOC, do inglês *Second Order Conditions*), de tal forma que é atingida uma convergência rápida para um mínimo local.

No seguimento destes trabalhos, os mesmos autores apresentam em [46] a implementação detalhada do algoritmo e os resultados numéricos obtidos.

Chin, seguindo o primeiro algoritmo de filtros apresentado por Fletcher e Leyffer, também efectuou em [8] uma análise teórica da convergência global de um algoritmo SQP de filtros com procura unidimensional.

Paralelamente, Antunes e Monteiro implementaram um algoritmo SQP de filtros com procura unidimensional em [1], onde apresentam os detalhes de implementação e alguns resultados numéricos obtidos.

Mais recentemente, em [28], Gould *et al.* introduzem um novo algoritmo para resolver problemas com equações e mínimos quadrados não lineares. Neste algoritmo combinam o método dos filtros, expandido para filtros multidimensionais, com a técnica de regiões de confiança.

Em [42], Ulbrich apresenta uma versão modificada do filtro-SQP com regiões de confiança de Fletcher *et al.* [15], prova a sua convergência local superlinear e mostra que os passos originais do filtro-SQP podem ser utilizados sem a adicional correcção de segunda ordem. Basicamente, a alteração introduzida neste algoritmo consiste na utilização do valor da função Lagrangeana no filtro em vez do valor da função objectivo em conjunto com medidas apropriadas de avaliação da admissibilidade. Também é mostrado que este algoritmo goza das mesmas propriedades de convergência global do algoritmo filtro-SQP original [15].

Em [25], Gould e Toint referem as linhas gerais do algoritmo FILTRANE, um algoritmo de filtros com regiões de confiança, apresentando os resultados numéricos obtidos. Este

algoritmo é analisado por Gould e Toint no documento [26], onde é apresentado um pacote *Fortran 95* para encontrar vectores que satisfaçam um conjunto de equações e/ou inequações não lineares. Também são discutidas variações do algoritmo e testados problemas do CUTEr, que indicam que tanto o algoritmo original como as variações são robustos e eficientes. Este documento mostra um primeiro estudo experimental dos parâmetros inerentes aos algoritmos dos filtros.

Neste levantamento de trabalhos que utilizam o método dos filtros, constatou-se que é usado com as duas técnicas de globalização - a procura unidimensional e as regiões de confiança. Pode ainda concluir-se que é utilizado na optimização com e sem restrições e ainda na optimização sem derivadas. Para concluir, Gould e Toint, em [29], num documento de reflexão sobre o estado de maturidade da optimização não linear, referem-se à abordagem dos filtros como uma ideia simples e poderosa, que traduz "*o progresso mais significativo nos últimos cinco anos, e que originou, em tão pouco tempo, o interesse de vários investigadores*".

Capítulo 4

Técnicas usadas no algoritmo

Neste capítulo apresenta-se a ideia subjacente à restauração inexacta e descreve-se a técnica de globalização da procura unidimensional num contexto do método dos filtros. Por fim é apresentado o algoritmo que serviu de base a este trabalho, conjugando todas estas ideias.

A abordagem dos filtros já tem alguns desenvolvimentos, essencialmente na programação não linear com restrições, usando várias combinações de técnicas bem conhecidas, tais como regiões de confiança, pontos interiores, métodos barreira, procura padrão, procura unidimensional e também restauração inexacta, como já foi descrito na Secção 3.2.

Neste trabalho foi implementado um algoritmo que combina a abordagens dos filtros com as técnicas de restauração inexacta e procura unidimensional, apoiado no estudo teórico de convergência efectuado por Gonzaga *et al.* [23]. A abordagem utilizada neste algoritmo traduz-se na utilização do método dos filtros apresentado por Fletcher e Leyffer [13] combinado com a técnica de restauração inexacta (IR) introduzida por Martínez [34], e largamente utilizada [18], [17], [31], em conjunto com a estratégia de procura unidimensional [38].

4.1 Abordagem IR

O ponto de vista da abordagem IR baseia-se no facto de que a admissibilidade é um factor muito importante do problema e por isso deve ser controlado de modo independente da optimalidade. Assim, os métodos baseados em IR consideram a admissibilidade e a optimalidade em fases diferentes numa mesma iteração.

Um problema bem conhecido dos métodos de admissibilidade é a sua incapacidade de seguir domínios muito curvados, o que provoca o cálculo de passos muito pequenos numa zona longe da solução. A metodologia IR tenta evitar este inconveniente, utilizando procedimentos que automaticamente diminuem a tolerância de inadmissibilidade à medida que o processo iterativo se aproxima da solução - no início do processo, são calculados passos maiores que vão adaptativamente sendo diminuídos.

Um factor essencial nesta metodologia é que existe total liberdade de escolha do algoritmo a usar em cada fase, podendo desta forma ser melhor exploradas as características de cada problema.

O tratamento independente das fases de admissibilidade e optimalidade na abordagem IR, foi explorado por Martínez em [32] sendo justificado pelo facto delas apresentarem características diferentes no ponto óptimo. Na maioria dos problemas práticos, uma solução admissível não óptima pode ser útil, enquanto que um ponto inadmissível que satisfaça as condições de optimalidade é completamente desnecessário. Em [32] é apresentado um algoritmo que em cada iteração procura, primeiro a redução de $\|c(x)\|_2$, e depois a redução da função Lagrangeana. O novo ponto calculado pela iteração pode ser aceite ou rejeitado, de acordo com o valor da função mérito que combina a admissibilidade com a optimalidade. Este algoritmo resolve problemas com restrições de igualdade e limites simples nas variáveis. Em [35], Martínez acrescenta ao algoritmo já apresentado o tratamento das restrições de desigualdade e prova a convergência global do novo método utilizando como função mérito a Langrangeana aumentada, mas no entanto, a função mérito utilizada não permite a estimação arbitrária dos multiplicadores de Lagrange.

Martínez e Pilotta, [34], introduzem um novo modelo para resolver problemas de pro-

gramação não linear. Neste algoritmo não são utilizadas variáveis de folga para lidar com as restrições de desigualdade. Cada iteração do método tem duas fases, primeiro fase de admissibilidade e depois a fase da optimalidade. No final da segunda fase o ponto resultante é comparado com o ponto actual, utilizando uma função mérito que combina a optimalidade com a admissibilidade. Esta função mérito inclui um parâmetro de penalidade que muda entre iterações consecutivas. A algoritmo é implementado e comparado com o algoritmo LANCELOT utilizando o conjunto de problemas *hard-spheres*.

Em [33], Martínez introduz um novo método de *Inexact-Restoration* para programação não linear, onde cada iteração do algoritmo principal é composta por duas fase. Na fase 1, a admissibilidade é melhorada de modo explícito e na fase 2 é melhorada a optimalidade numa aproximação tangencial das restrições. São utilizadas regiões de confiança na redução do passo quando o novo ponto não é suficientemente bom. Nesta abordagem, o ponto intermédio mais admissível é considerado melhor do que o ponto actual. Este é o primeiro método em que um ponto intermédio centrado em regiões de confiança é combinado com o decréscimo da função Lagrangeana numa aproximação tangencial das restrições. A função mérito utilizada neste documento também é nova: consiste numa combinação convexa da Lagrangeana com a norma das restrições. É provada a convergência global, apresentado e justificado teoricamente um algoritmo para a primeira fase e efectuados alguns testes numéricos.

O método IR tem sido alvo de interesse de vários investigadores, em áreas distintas, pois é uma estratégia que pode ser facilmente combinada com outras técnicas. Birgin *et al.* [5] desenvolveram um algoritmo para processos químicos de produção e outros problemas de engenharia. São apresentados exemplos numéricos da utilização do algoritmo. Friedlander e Castro [11] propõem uma nova abordagem de resolução de problemas de programação matemática a dois níveis. Esta abordagem permite a resolução directa do problema de segundo nível, sem reformulação, e sem necessidade de recorrer a técnicas de optimização não diferenciável. Isto é possibilitado pela utilização do algoritmo de *Inexact-Restoration* apresentado por Martínez e Pilotta. Andreani *et al.*, em [39], apresentam um algoritmo de resolução de problemas de congestionamento urbano utilizando a abordagem de *Inexact-*

Restoration.

Em [31], Martínez e Pilotta apresentam as principais ideias subjacentes ao método *Inexact-Restoration*, assim como avanços mais recentes e aplicações do método. Salientam o facto de já existirem disponíveis diferentes algoritmos IR em que os mais recentes utilizam a abordagens das regiões de confiança. Também que a convergência global já foi provada baseada em funções do tipo Lagrangeana aumentada.

Mais recentemente, Birgin e Martínez em [18], provam a convergência local de um algoritmo de *Inexact-Restoration* para programação não linear. Efectuam experiências numéricas com o objectivo de avaliar o comportamento de métodos puramente locais contra a convergência global de algoritmos de programação não linear.

4.2 Procura Unidimensional com Filtros

Para dar uma ideia acerca do funcionamento desta técnica e por uma questão de simplificação, apresenta-se a técnica de procura unidimensional, num contexto de optimização sem restrições, *i.e.*, em que o único objectivo é minimizar $f(x)$.

A maioria dos métodos de optimização possuem uma convergência do tipo local, ou seja, convergem para a solução se a estimativa inicial do processo iterativo estiver suficientemente próxima da solução. Nos métodos que convergem, independentemente da aproximação inicial x^0, a convergência diz-se global e os métodos globalmente convergentes. Uma vez que raramente se tem informação acerca da solução do problema, a selecção duma estimativa inicial na região de convergência, não é uma tarefa fácil. Surge então a necessidade de introduzir alterações aos métodos localmente convergentes, com o objectivo de alargar a região de convergência, tornando-os globalmente convergentes, mantendo no entanto as suas excelentes propriedades de convergência local. As alterações referidas são identificadas como técnicas de globalização.

Existem dois esquemas fundamentais de indução de convergência global - procura unidimensional e regiões de confiança. Ambas as técnicas geram passos de procura, cada uma à sua maneira. A procura unidimensional usa um modelo para gerar a direcção de procura

e em seguida tenta encontrar o comprimento do passo adequado ao longo dessa direcção. A técnica de regiões de confiança define uma região à volta da aproximação actual, dentro da qual o modelo quadrático parece ser uma representação adequada da função objectivo, escolhendo como passo o minimizante do modelo nessa região de confiança. Na prática a direcção e o comprimento do passo são calculados em simultâneo. Se um passo não pode ser aceite, o tamanho da região de confiança é reduzido e a técnica procura um novo minimizante. Em geral, a direcção muda sempre que o tamanho da região de confiança é alterado.

Estas duas técnicas de globalização de métodos são usadas com sucesso, em algoritmos de optimização, e nenhuma delas se tem mostrado superior à outra. Experiências computacionais têm mostrado que, em média, elas são comparáveis em termos de eficiência, apesar de, em determinados problemas o seu desempenho ser diferente.

O termo procura unidimensional surge devido ao facto da procura ser baseada numa função de apenas uma só variável. Idealmente, o objectivo seria calcular a solução de

$$\arg\ min\, \varphi(\alpha) \equiv f(x^k + \alpha d^k), \tag{4.1}$$

ou seja, α seria o minimizante global da função a uma variável $\varphi(\alpha)$. Este problema, mesmo para o cálculo dum minimizante local, quando resolvido exactamente torna-se nalguns casos inviável e pode ser muito pesado em termos de custos computacionais. Na prática aceita-se uma aproximação a (4.1) de tal forma que se verifique uma redução no valor da função objectivo com o mínimo de esforço computacional. Esta estratégia caracteriza uma procura monótona. Os algoritmos típicos de procura unidimensional experimentam uma sucessão de possíveis valores para α aceitando o primeiro que satisfaça uma determinada condição de monotonia da função objectivo. O sucesso desta técnica depende das escolhas feitas para a direcção e para o comprimento do passo.

Como já foi referido, a convergência global pode ser forçada através de esquemas de procura unidimensional que obriguem a uma redução significativa de determinada função. No entanto, quando se está muito próximo da solução x^*, os esquemas de imposição da redução da função objectivo podem obrigar a que não seja aceite um comprimento de passo

unitário ($\alpha = 1$), pondo em causa a razão de convergência local do método. Esta situação foi apresentada pela primeira vez por Maratos na sua tese de doutoramento, ficando a ser conhecida pelo "efeito de Maratos".

Neste trabalho, passando agora para a optimização com restrições, a utilização desta técnica de globalização foi feita conjuntamente com um esquema baseado no método dos filtros. Assim, o objectivo é determinar o valor de α, para que o próximo ponto do processo

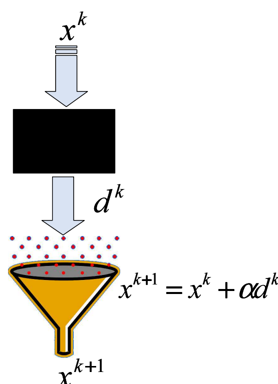

Figura 4.1: Esquema da procura unidimensional com filtros

iterativo, $x^{k+1} = x^k + \alpha\, d^k$, origine um par $(f(x^{k+1}), h(x^{k+1}))$ aceite pelo filtro. Começa-se com um valor inicial para α e, caso o novo ponto não seja aceite pelo filtro, α é dividido por dois, o ponto x^{k+1} é actualizado e verificado novamente pelo filtro, sem que seja resolvido mais nenhum problema.

Algoritmo 4.2.1 *Procura Unidimensional*

Inicializar $\alpha = 1$;
REPETIR
  Calcular a direcção d^k;
  Actualizar o ponto: $x^{k+1} = x^k + \alpha\, d^k$;
  Se *o par* $(f(x^{k+1}), h(x^{k+1}))$ *é aceite pelo filtro* ***Então***
    Aceitar o ponto x^{k+1};
  Senão
    Rejeitar o ponto x^{k+1};
    $\alpha = \dfrac{\alpha}{2}$;
ATÉ *Aceitar* x^{k+1} ***ou*** $\alpha \leq TolAlpha$.

Ao contrário da optimização sem restrições, não é obrigatório que o valor da função objectivo $f(x)$ decresça em todas as iterações do processo iterativo. Esta avaliação do novo ponto é feita, tendo em conta os dois objectivos - minimização da função objectivo e minimização da violação das restrições (ver Secção 3.1). O filtro, funciona como um "oráculo".

O valor de α deverá produzir uma redução significativa no valor da função objectivo ou na violação das restrições e não deverá ser demasiado pequeno. Em termos do conceito de filtro, esta exigência reflecte-se no envelope apresentado anteriormente na Figura 3.8.

4.3 Algoritmo base

O algoritmo implementado neste trabalho foi introduzido por Gonzaga *et al.* [23], onde apenas é apresentado o algoritmo principal, sem especificação dos algoritmos internos utilizados em cada uma das fases, nem a forma como o filtro é gerido. O trabalho destes autores consiste num estudo teórico de um algoritmo de filtros utilizando a abordagem IR. Também é apresentada a convergência global do algoritmo.

Os dados iniciais do algoritmo são o ponto inicial x^0, o filtro inicial, para definir os pontos da região de aceitação, $\mathcal{F}_0 = \emptyset$, no início vazio, e o filtro que define os pontos da região de rejeição, $\digamma_0 = \emptyset$, no inicio vazio. Este algoritmo pode ser visto em três grandes blocos de cálculo. No primeiro bloco, correspondente à fase da admissibilidade, começa-se por calcular um par $(f(x^k), h(x^k))$, correspondente ao ponto inicial, e insere-se temporariamente este par no filtro $\mathcal{F}_k$. De seguida é determinado um ponto intermédio, "mais admissível", z^k, a partir do ponto x^k; no segundo bloco, da fase da optimalidade, é calculado um ponto x^{k+1}, com melhor valor de f, a partir do ponto intermédio z^k; por fim, no último bloco decide-se se o par temporário inicial $(f(x^k), h(x^k))$ é mantido no filtro ou se é removido. O par permanece no filtro sempre que o novo ponto, x^{k+1} tem um valor da função objectivo $f(x^{k+1})$ significativamente maior do que o valor da função objectivo $f(x^k)$ do ponto inicial.

Algoritmo 4.3.1 *Algoritmo Base*

Dados x^0, λ^0, $\mathcal{F}_0 = \emptyset$ *é o filtro que define a região de aceitação inicial e*
$\Gamma_0 = \emptyset$ *o filtro que define a região de rejeição inicial*
$k = 0$
REPETIR
  Fase da Admissibilidade:
    Introduzir o par $(f(x^k), h(x^k))$ *temporariamente no filtro* $\mathcal{F}_k$;
    Determinar z^k.
  Se z^k *verificar o Critério de Paragem* ***Então*** *Termina com sucesso.*
  Senão
    Fase da Optimalidade:
      Determinar x^{k+1}.
    Actualização do Filtro:
      Se $f(x^{k+1}) \geq f(x^k) - min(h(x^k)^2, \omega)$ ***Então***
        Mantém o par temporário $(f(x^k), h(x^k))$ *no filtro e remove os pares dominados;*
        Remove o par temporário $(f(x^k), h(x^k))$ *do filtro.*
  $k = k + 1$
  $x^k = x^{k+1}$
ATÉ x^k *verificar o Critério de Paragem.*

O algoritmo implementa a técnica IR, esquematizada na Figura 4.2, com duas fases independentes - a fase da admissibilidade, seguida da fase da optimalidade. Note-se que em cada uma das fases são resolvidos problemas diferentes, com funções objectivo e restrições distintas, no entanto, em ambas as fases é utilizado o método dos filtros combinado com a técnica da procura unidimensional.

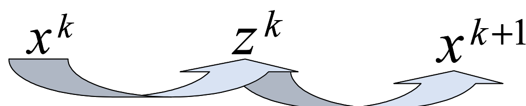

Figura 4.2: Técnica IR

Os autores provam convergência global para pontos estacionários e referem que o método é independente dos algoritmos internos usados em cada iteração, desde que estes

algoritmos satisfaçam alguns requisitos na sua eficiência. Mostram que, sob certas hipóteses, para um filtro com tamanho mínimo, o algoritmo gera um ponto de acumulação estacionário e que para um filtro de maior dimensão, todos os pontos de acumulação são estacionários. Sugerem a introdução de esquemas de correcção de segunda ordem no algoritmo base, por forma a evitar o efeito de Maratos.

As hipóteses *standard* utilizadas são:

- H1: os iterandos x^k e z^k permanecem num domínio convexo e compacto $X \subset \mathbb{R}^n$
- H2: todas as funções (objectivo e restrições) são continuamente diferenciáveis à Lipschitz num conjunto aberto contendo X.
- H3: todos os pontos de acumulação admissíveis $\overline{x} \in X$ de x^k, satisfazem a condição de Mangasarian- Fromovitz (MPQC), *i.e.*, os gradientes das restrições de igualdade são linearmente independentes, e existe uma direcção $d \in \mathbb{R}^n$ tal que $A_E(\overline{x})d = 0$ e $A_{\overline{I}}(\overline{x}) < 0$ em que $\overline{I} = \{i \in I |\, c_i(\overline{x}) = 0\}$.

No próximo capítulo vai ser feita a apresentação do algoritmo desenvolvido que tem como base o Algoritmo 4.3.1.

Capítulo 5

Algoritmo IR com Filtros

Este capítulo é vocacionado para os detalhes de implementação - são apresentados os algoritmos das fases de admissibilidade e optimalidade. São referidas as características de gestão do filtro, as opções tomadas relativamente a alguns procedimentos e ainda mostrados detalhes da interface do código desenvolvido com a linguagem de modelação AMPL.

Foi apresentado no capítulo anterior, o algoritmo base proposto por Gonzaga *et al.* em [23] e que serviu de motivação para este trabalho. O algoritmo final é designado por Algoritmo IR com Filtros. Os procedimentos internos a cada uma das fases da abordagem IR não são especificados em [23], sendo este o capítulo dedicado a esse fim.

Neste trabalho apenas são tratados problemas com restrições do tipo desigualdade.

5.1 Componentes do algoritmo

5.1.1 Admissibilidade

Esta fase do algoritmo apenas se destina ao problema da admissibilidade, não sendo realizada se o ponto x^k já for, à partida, admissível.

É efectuada uma aproximação linear da violação das restrições - o objectivo consiste em calcular uma direcção de procura de forma a encaminhar o processo iterativo para uma região em que a violação das restrições é menor. Pretende-se minimizar a violação das

restrições, satisfazendo sempre as restrições que já são verificadas. Assim, é resolvido o seguinte problema linear (LP, do inglês *Linear Programming*):

$$\begin{aligned} &\min_{{d_{fea}}^k \in \mathbb{R}^n} \quad \sum_{i \in J} A_i^k {d_{fea}}^k \\ &s.a. \qquad A_i^k {d_{fea}}^k + c_i^k \leq 0, \qquad i \in \mathbb{R}^n J^{\perp} \end{aligned} \tag{LP}$$

em que $A^k = \nabla c(x^k)^T$ representa a matriz do Jacobiano das restrições e, J e $J^{\perp}$ representam os conjuntos das restrições violadas e satisfeitas, respectivamente. A direcção de procura ${d_{fea}}^k$ (fea, do inglês *feasibility*), é a solução do problema (LP), cuja resolução é também um procedimento iterativo correspondente à primeira fase da Figura 4.2. O ponto $z^k = x^k + \alpha\ {d_{fea}}^k$ é obtido pela técnica da procura unidimensional utilizando o filtro, como foi descrito na Secção 4.2, onde $\alpha \in \mathbb{R}$ representa o tamanho do passo.

O ponto intermédio z^k é aceite se o par correspondente $(f(z^k), h(z^k))$ for aceite pelo filtro, isto é, não for dominado por nenhum outro do filtro. As condições de decréscimo significativo para aceitação do filtro, definidas em (3.2) e (3.3), têm de ser verificadas no ponto z^k:

$$h(z^k) < \beta h^l \quad \text{ou} \quad f(z^k) < f^l - \gamma h(z^k). \tag{5.1}$$

Além da aceitação pelo filtro, o ponto intermédio z^k também tem de produzir um decréscimo significativo na violação das restrições, quando comparado com o ponto temporário x^k, para que seja aceite. Assim, a condição seguinte tem de ser verificada:

$$h(z^k) \leq (1 - \delta) h(x^k) \tag{5.2}$$

onde δ é uma constante positiva perto de zero.

Se alguma destas condições (5.1) e (5.2) não se verificar, o ponto z^k é rejeitado e α é divido por dois até que o ponto seja aceite ou até que α seja menor do que uma determinada tolerância (Figura 4.1).

Se o ponto intermédio z^k verificar o critério de paragem, o algoritmo termina de imediato com sucesso. Caso contrário, passa para a fase seguinte, a fase da optimalidade.

5.1.2 Optimalidade

O objectivo desta a fase é, a partir de um ponto "mais admissível" z^k, reduzir a função objectivo. Nesta fase, a partir de z^k, o problema a resolver tem como função objectivo uma aproximação quadrática de $f(x)$ e como restrições aproximações lineares de $c(x)$. Assim, é resolvido o seguinte subproblema quadrático (QP, do inglês *Quadratic Programming*), uma aproximação ao problema original (NLP):

$$\begin{array}{ll} \underset{{d_{opt}}^k \in \mathbb{R}^n}{min} & \frac{1}{2}({d_{opt}}^k)^T W^k {d_{opt}}^k + ({d_{opt}}^k)^T g^k \\ s.a. & A^k {d_{opt}}^k + c^k \leq 0 \end{array} \tag{QP}$$

onde $g^k = \nabla f(z^k)$ é o gradiente da função objectivo, $A^k = \nabla c(z^k)^T$ é a matriz do Jacobiano das restrições $c(z^k)$ e $W^k = \nabla^2 L(z^k)$ é a matriz Hessiana da função Lagrangeana de (NLP).

A solução do subproblema (QP) é a direcção de procura ${d_{opt}}^k$. O algoritmo de cálculo desta direcção é também um procedimento iterativo. Esta direcção de procura vai determinar o novo ponto $x^{k+1} = z^k + \alpha\ {d_{opt}}^k$. A seguir é avaliada a aceitação do par $(f(x^{k+1}), h(x^{k+1}))$ pelo filtro, *i.e.*, a verificação de (5.1) e (5.2) em x^{k+1}. Nesta fase, também são ainda comparadas as funções objectivo no novo ponto $f(x^{k+1})$ e no ponto intermédio $f(z^k)$:

$$f(x^{k+1}) \leq (1-\delta) f(z^k) \tag{5.3}$$

onde δ é uma constante positiva perto de zero.

Se o par $(f(x^{k+1}), h(x^{k+1}))$ for aceite pelo filtro e se a condição (5.3) se verificar, x^{k+1} é a aproximação seguinte a ser avaliada pelo algoritmo. Caso contrário, se uma das condições não se verificar, o α é dividido por dois até que o ponto x^{k+1} seja aceite ou até que α seja menor do que uma determinada tolerância (processo idêntico ao da fase da admissibilidade). Se o ponto x^{k+1} verificar o critério de paragem, o algoritmo termina com sucesso senão é feita a actualização do filtro, descrita a seguir.

5.1.3 Actualização do filtro

Na gestão do filtro existem dois procedimentos base - a verificação de aceitação de um ponto no filtro e a sua inserção. O primeiro deles, o de aceitação de um ponto no filtro, é igual ao introduzido por Fletcher e Leyffer, referido na Secção 3.1, do Capítulo 3, sendo utilizado na fase de admissibilidade e optimalidade. Relativamente ao esquema de inserção de um ponto no filtro, foram introduzidas algumas particularidades - existe um ponto temporário x^k que só é inserido no filtro, se for aceite por este e, quando comparado com o último ponto do processo iterativo, x^{k+1}, verificar a seguinte condição:

$$f(x^{k+1}) \geq f(x^k) - min(h(x^k)^2, \omega) \tag{5.4}$$

em que ω, é uma constante pequena positiva. Esta condição apresenta alguma exigência e tem como objectivo tentar encontrar um ponto ainda melhor sem o introduzir no filtro, evitando aumentar o número de pares no filtro. Se for verdadeira, então averigua-se também se o ponto temporário é aceite pelo filtro. Se estas duas condições se verificarem, o par temporário $(f(x^k), h(x^k))$ é inserido no filtro e são removidos os pares dominados. Note-se que este procedimento vai permitir que apenas seja introduzido no filtro o ponto temporário x^k, quando o novo ponto x^{k+1} tiver maior valor de f do que x^k, originando a inserção de um conjunto mais restrito de pontos, tornando assim, o filtro mais pequeno e a sua gestão facilitada. Finalmente, averigua-se se o novo ponto x^{k+1} verifica o critério de paragem e em caso afirmativo o algoritmo termina com sucesso, caso contrário uma nova iteração é realizada.

5.1.4 Critério de paragem

O algoritmo termina quando for encontrado um ponto, x^*, que verifique as seguintes condições: o ponto tem de ser estacionário (KKT) e tem de verificar todas as restrições, isto é, o valor de h no ponto tem de ser suficientemente pequeno. Foi necessário introduzir esta segunda condição no critério de paragem pois verificou-se que em algumas situações raras, o algoritmo converge para um ponto estacionário violando algumas restrições. Este

facto foi apontado pelos autores no seu estudo teórico da convergência ([23]) como sendo possível de acontecer em determinadas situações.

A primeira condição do critério de paragem diz respeito à verificação da estacionaridade (cálculo do resíduo normalizado r) e a segunda à análise da admissibilidade:

$$r(x^*) = \frac{\|g^* + \nu^* + A^* \lambda^*\|_2}{\max\{\mu_{\max}, 1.0\}} \leq \epsilon \qquad \text{e} \qquad h(x^*) \leq \xi. \tag{5.5}$$

em que $g^* = \nabla f(x^*)$, $A^* = \nabla c(x^*)$, ν e λ são os vectores dos multiplicadores de Lagrange das restrições de limites simples e das restantes restrições, respectivamente, ϵ é uma constante pequena e positiva e $\mu_{\max} = \max_i \{\|g^*\|_2, |\nu_i|, \|a_i^*\|_2 |\lambda_i^*|\}$.

5.2 Algoritmo final

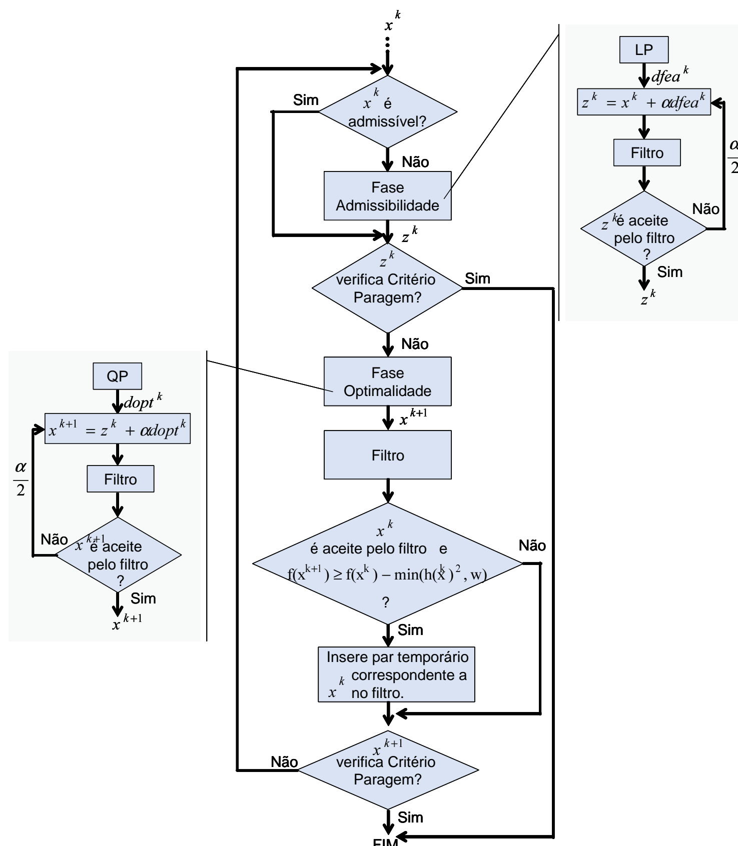

Figura 5.1: Fluxograma do Algoritmo IR com Filtros

É apresentado um esquema do algoritmo no fluxograma da Figura 5.1. O algoritmo, na primeira iteração, calcula o par temporário $(f(x^0), h(x^0))$ correspondente ao ponto inicial x^0 e inicializa o filtro $\mathcal{F}^0$ com $(-\infty, +\infty)$. O par temporário não é logo inserido no filtro, esta decisão terá lugar no último bloco do algoritmo - este facto, como já foi referido, permite reduzir significativamente o número de inserções no filtro.

Algoritmo 5.2.1 *Algoritmo IR com Filtros*

Dados x^0, λ^0, $\mathcal{F}^0$ *é o filtro inicial e* $(f(x^0), h(x^0))$ *é o par temporário;* $k = 0$
REPETIR
  Fase da Admissibilidade:
    Determinar z^k*:*
      Resolver subproblema (LP) em x^k *para determinar* ${d_{fea}}^k$*;*
      $\alpha = 1$;
      REPETIR
        $z^k = x^k + \alpha\, {d_{fea}}^k$;
        Se $(f(z^k), h(z^k))$ *é aceite pelo filtro* ***E*** $h(z^k) \leq (1-\delta)h(x^k)$ ***Então***
          Aceitar z^k*;*
        Senão
          Rejeitar z^k*;*
          $\alpha = \dfrac{\alpha}{2}$;
      ATÉ *Aceitar* z^k ***ou*** $\alpha \leq TolAlpha$*;*
  Se z^k *verificar o Critério de Paragem* ***Então*** *Termina com sucesso.*
  Senão
    Fase da Optimalidade:
      Determinar x^{k+1}*:*
        Resolver subproblema (QP) em z^k *para determinar* ${d_{opt}}^k$*;*
        $\alpha = 1$;
        REPETIR
          $x^{k+1} = z^k + \alpha\, {d_{opt}}^k$;
          Se $(f(x^{k+1}), h(x^{k+1}))$ *é aceite pelo filtro* ***E*** $f(x^{k+1}) \leq (1-\delta)f(z^k)$ ***Então***
            Aceitar x^{k+1}*;*
          Senão
            Rejeitar x^{k+1}*;*
            $\alpha = \dfrac{\alpha}{2}$;
        ATÉ *Aceitar* x^{k+1} ***ou*** $\alpha \leq TolAlpha$*;*
    Actualização do Filtro:
      Se $f(x^{k+1}) \geq f(x^k) - min(h(x^k)^2, \omega)$ ***Então***
        Se $(f(x^k), h(x^k))$ *é aceite pelo filtro* ***Então***
          Insere o par temporário $(f(x^k), h(x^k))$ *no filtro;*
          Remove os pares dominados por $(f(x^k), h(x^k))$ *do filtro;*
  $k = k + 1$;
  $x^k = x^{k+1}$;
ATÉ x^k *verificar o Critério de Paragem.*

5.2.1 Exemplo ilustrativo

Analise-se um exemplo para melhor explicar o funcionamento do algortimo.

Considere-se a Figura 5.2 onde estão representados alguns pares num filtro, definindo a região de aceitação e a região proibida. Considere-se também o ponto a azul como um ponto inicial temporário. Este ponto encontra-se na região de aceitação do filtro, mas ainda viola algumas das restrições, por isso, será executada a fase da admissibilidade.

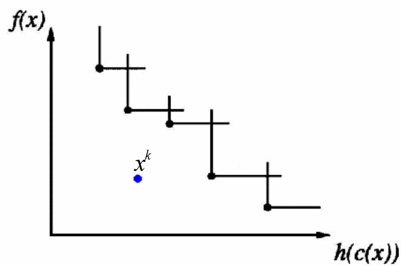

Figura 5.2: Ponto temporário inicial x^k

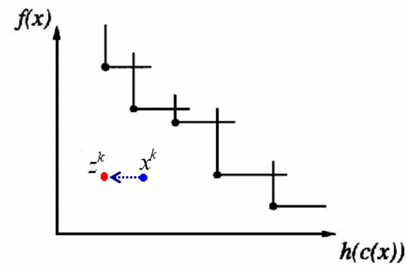

Figura 5.3: Admissibilidade - ponto intermédio z^k

Da admissibilidade resulta o ponto z^k, representado a vermelho na Figura 5.3. O ponto z^k não verificando o critério de paragem, obriga à execução da fase de optimalidade, determinando-se o ponto x^{k+1}, a partir de z^k, como mostra a Figura 5.4.

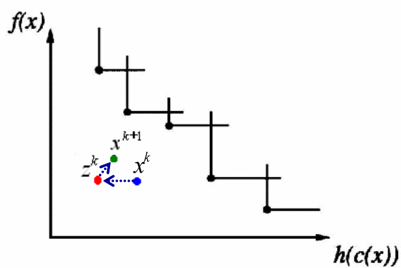

Figura 5.4: Optimalidade - nova aproximação x^{k+1}

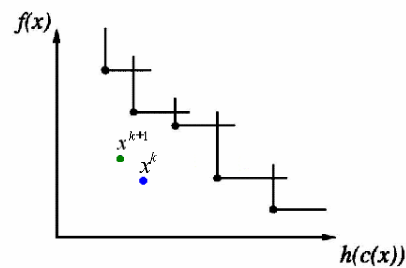

Figura 5.5: Análise da necessidade de inserir o ponto intermédio x^k no filtro

Finalmente, falta averiguar se o ponto temporário x^k é inserido ou não no filtro. Na Figura 5.5 verifica-se que o ponto x^k se encontra na região de aceitação do filtro, e que

o ponto x^{k+1}, a verde, tem pior valor de f do que o ponto x^k, a azul. Logo, o ponto temporário x^k é inserido no filtro (Figura 5.6). Ao inserir este ponto no filtro verifica-se pela Figura 5.7 que este domina dois pontos já lá existentes que serão removidos.

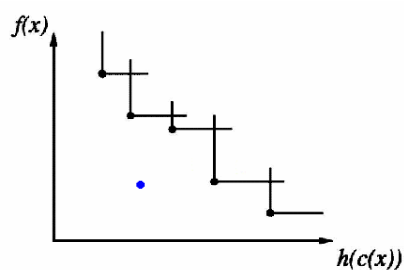

Figura 5.6: Inserção do ponto intermédio x^k no filtro

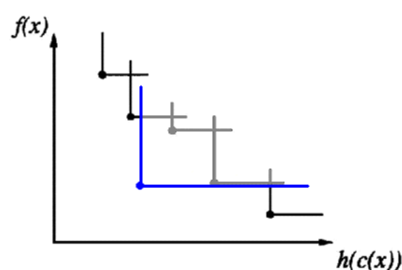

Figura 5.7: Remoção dos pontos dominados

Após a inserção do ponto x^k e respectiva remoção dos pontos dominados do filtro, este fica com a configuração apresentada na Figura 5.8. O filtro final é apresentado na Figura

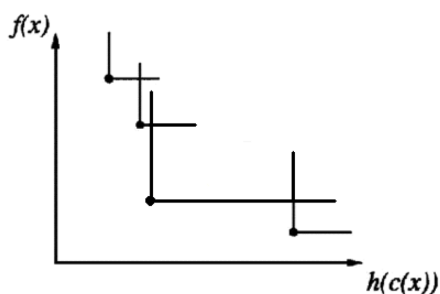

Figura 5.8: Filtro final

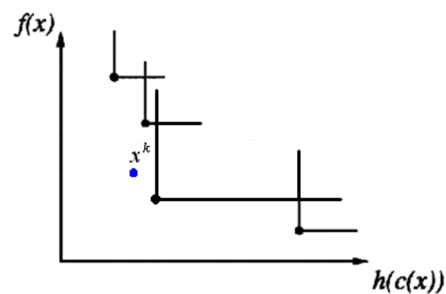

Figura 5.9: Filtro final e novo ponto temporário

5.9. A nova aproximação x^{k+1} será o novo ponto temporário a ser testado na próxima iteração.

5.3 Detalhes da implementação

O algoritmo implementado resolve problemas codificados na Linguagem de Modelação de Problemas Matemáticos AMPL (do inglês *Modelling Language for Mathematical Programming*) [16]. Apresenta-se, de seguida, um exemplo de um problema escrito nesta linguagem.

Exemplo de um problema em AMPL: *hs001.mod*

```
var x {1..2}; minimize obj: 100*(x[2] - x[1]^2)^2 + (1-x[1])^2;
subject to constr: -1.5 <= x[2];
let x[1] := -2;
let x[2] := -2;
```

A primeira fase da implementação consiste na ligação do mesmo à linguagem AMPL por forma a permitir a leitura dos dados necessários à execução do algoritmo ([19]).

O processo de interface com a linguagem AMPL trata-se, na generalidade, da leitura de um ficheiro com extensão *mod* para determinadas estruturas passíveis de serem utilizadas pelo algoritmo implementado na linguagem de programação C. Para que o ficheiro escrito na linguagem AMPL possa ser interpretado pelo algoritmo é necessário que este seja convertido, pelo próprio AMPL, num ficheiro intermédio, ficheiro *Stub.nl*, um formato não editável, apenas utilizado no carregamento das estruturas para o algoritmo, tal como esquematizado pela Figura 5.10.

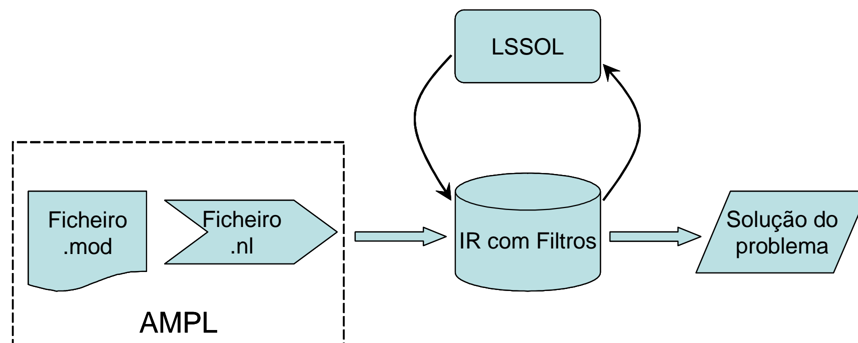

Figura 5.10: Esquema do Algoritmo IR com Filtros com a interface do AMPL e com a subrotina LSSOL

A interface com o AMPL é possibilitada por um conjunto de funções desenvolvidas para o efeito e compiladas em bibliotecas específicas, tal como é explicado no seu relatório

técnico [19]. Das bibliotecas disponíveis foram utilizadas duas, invocadas pelo seguintes comandos:

```
#include "amplinc\nlp.h"
#include "amplinc\getstub.h"
```

Assim, na implementação do algoritmo foi definida uma estrutura que vai armazenar todos os componentes do ficheiro *.nl*, declarada do seguinte modo:

```
#define asl cur_ASL ASL *asl;
```

A instrução seguinte define que se pretende a informação sobre as primeiras e segundas derivadas:

```
asl = ASL_alloc(ASL_read_pfgh);
```

A função *getstops* devolve o ficheiro AMPL a ler com as opções especificadas na estrutura *Oinfo*.

```
struct Option_Info Oinfo = { "filterphased", "FILTERPHASED",
"filter_options", keywds, nkeywds, 1, filter_version, 0, NULL};

stub = getstops(argv, &Oinfo);
```

O carregamento propriamente dito do ficheiro *.nl* é efectuado pela função *jac_ dim_ ASL*, da seguinte forma:

```
 nl=jac_dim_ASL(asl, stub, &m, &n, &no, &nz,
&mxr, &mxc, (fint) strlen(stub));
```

Esta função lê a primeira parte do ficheiro e armazena nas variáveis, passadas como argumentos, os seguintes valores, entre outros:

- m : representa o número de restrições
- n : representa o número de variáveis
- no : define o número de funções objectivo

Finalmente, a leitura do ficheiro *.nl* é terminada com a invocação da instrução:

```
pfgh_read_ASL(asl, nl, 0);
```

Os limites inferiores e superiores das restrições de limites simples e das $c(x)$ são guardados em dois vectores, *lb* e *ub*, respectivamente. O preenchimento destes vectores é efectuado pelas instruções seguintes:

```
/*LUv -> vector que alterna limites simples inferiores e
superiores das variaveis - n variaveis */
tmp=LUv;
for(i=0;i<n;i++){
    *lbt++=*tmp++;
    *ubt++=*tmp++;
}

/*LUrhs -> vector que alterna limites simples inferiores e
superiores das restricoes - m restricoes */
tmp=LUrhs;
for(i=0;i<m;i++){
    *lbt++=*tmp++;
    *ubt++=*tmp++;
}
```

Na iteração k, para o valor de x^k, a função *jacval* determina a matriz do Jacobiano das restrições, *objgrd* calcula o gradiente da função objectivo, *conval* avalia o valor das restrições e guarda-o no vector *constx* e a função *objval* calcula o valor da função objectivo.

```
jacval(xk, jacobian, NULL);

objgrd(0, xk, gradObj, NULL);

conval(xk, constx, NULL);

fxk = objsign*objval(0, xk, NULL);
```

A fase da admissibilidade resolve um problema (LP), enunciado na Subsecção 5.1.1. Assim, é invocada a subrotina LSSOL do NPSOL [20] para resolver o problema (LP) que

tem como solução a direcção de procura d_{fea}. A invocação desta subrotina é efectuada pela seguinte instrução:

```
lsoptn_("Problem Type LP", 15); //Problema do tipo LP

lsoptn_("Feasibility tolerance = 1.0e-12", 31);

lssol_(&M, &n, &NCLIN, &NROWC, &NROWA, jacobian, BL, BU, CVEC,
ISTATE, KX, dfea, A, B, INFORM, &ITER, &OBJ, CLAMBDA, IW, &LENIW,
W, &LENW);
```

Dos parâmetros desta subrotina salientam-se os seguintes:

- BL : vector com os limites inferiores de todas as restrições
- BU: vector com os limites superiores de todas as restrições
- CVEC : vector com os coeficientes das N variáveis da função objectivo
- CLAMBDA : vector λ estimado pela subrotina (parâmetro de saída)
- dfea : direcção de procura, solução do subproblema (LP) (parâmetro de saída)
- INFORM: variável que informa se o problema foi resolvido com sucesso ou insucesso (parâmetro de saída)

O vector *CVEC* é o somatório das n componentes da matriz do Jacobiano das restrições violadas. Os vectores *BL* e *BU* são preenchidos com o valor das restrições que já são satisfeitas, sendo as restrições violadas, colocadas no vector *CVEC*, como livres. A colocação de uma restrição livre corresponde a atribuir ao limite violado o valor de ∞. No entanto, esta solução apresentou uma particularidade que necessitava de tratamento especial - quando todas as restrições são violadas, o problema fica sem restrições. Assim, neste caso a solução adoptada foi seleccionar apenas a restrição com maior violação para incorporar a função objectivo, ficando as restantes como restrições deste problema. Na situação de todas as restrições serem violadas, ainda existe um caso particular que é quando o problema tem

apenas uma restrição. Neste caso, a opção adoptada foi passar de imediato para a fase de optimalidade.

No final da fase da admissibilidade é feita a procura unidimensional com filtros para calcular o passo de procura α. Este valor é inicializado a 1, se o ponto $x^k + \alpha d_{fea}$ não for aceite pelo filtro, o valor de α é dividido por dois e o novo ponto é testado. Este processo é repetido até que o ponto seja aceite ou o α seja menor do que uma tolerância. Se o valor de α atingir a tolerância, decidiu-se utilizar o valor inicial de $\alpha = 1$ para o cálculo do ponto intermédio - $z^k = x^k + d_{fea}$. Embora este novo ponto não seja aceite pelo filtro, verificou-se que na grande maioria dos problemas o processo reencaminhava-se no sentido da solução do problema. Em alguns casos, estas sucessivas divisões de α estão relacionadas com o efeito de Maratos.

A fase da optimalidade resolve um subproblema completamente diferente da fase anterior. Aqui, como já referido, o objectivo é determinar um direcção de procura que conduza a um ponto com menor valor de f. Deste modo, é resolvido o subproblema (QP) recorrendo à subrotina LSSOL, parametrizada para resolver problemas do tipo (QP). A invocação da subrotina LSSOL para resolver este problema tem a seguinte configuração:

```
lsoptn_("Problem Type QP2", 16); // Problema do tipo QP2
lssol_(&n, &n, &NCLIN, &NROWC, &NROWA, jac, QPlb, QPub, gradObj,
ISTATE, KX, dopt, hess, B, INFORM, &ITER, &OBJ, CLAMBDA, IW,
&LENIW, W, &LENW);
```

Dos parâmetros desta subrotina salientam-se os seguintes:

- QPlb : vector com os limites inferiores de todas as restrições
- QPub: vector com os limites superiores de todas as restrições
- gradObj : vector do gradiente da função objectivo (função objectivo do problema a resolver)
- hess : matriz Hessiana da Lagrangeana

- CLAMBDA : vector λ estimado pela subrotina (parâmetro de saída)
- dopt : direcção de procura, solução do subproblema (QP) (parâmetro de saída)

Para calcular a matriz Hessiana da função Lagrangeana, parâmetro de entrada da subrotina LSSOL para problemas do tipo (QP), é necessária a estimação do vector dos multiplicadores de Lagrange λ. Este valor também é necessário para o cálculo do resíduo no critério de paragem. Na primeira iteração, utilizou-se para estimação deste vector o vector unitário. Para as iterações seguintes utilizou-se o valor fornecido pela rotina do LSSOL na resolução do problema (QP) (CLAMBDA).

O cálculo da matriz Hessiana da função Lagrangeana é efectuada pela seguinte invocação da função *fullhes* da biblioteca do AMPL:

```
fullhes(hessianLagr,n,0,NULL,lambda);
```

Capítulo 6

Resultados Computacionais

Neste capítulo são apresentadas as experiências computacionais realizadas. Foram seleccionados 235 problemas em AMPL sendo testadas duas versões do algoritmo base - a primeira versão com a fase de admissibilidade e a fase da optimalidade, a segunda com apenas a fase da optimalidade. Resolveu-se o mesmo conjunto de problemas com os softwares comerciais LOQO e NPSOL. É feita uma análise dos resultados baseada nos gráficos de perfil de desempenho de Dolan e Moré.

6.1 Ambiente de programação e testes numéricos

As experiências computacionais foram realizadas num *centrino* com 504MB de RAM. O algoritmo foi implementado na linguagem C no sistema operativo Windows. Os problemas (LP) e (QP) foram resolvidos pela subrotina LSSOL [20] do *software* NPSOL [21].

Os testes computacionais foram efectuados em etapas - numa primeira etapa foram testados 235 problemas com o Algoritmo IR desenvolvido. Estes resultados estão registados na Tabela 6.1. Esta tabela apresenta o nome do problema, a sua dimensão (n e m são o número de variáveis e de restrições, respectivamente), $\#it$, $\#f$ e $\#g$ representam o número de iterações, de cálculos de função e de cálculos de gradiente, respectivamente. A coluna $\#insfiltro$ designa o número de inserções no filtro e a coluna $\#elemfiltro$ contém a dimensão do filtro no final do algoritmo. Existe ainda uma coluna para observações relevantes (obs). A coluna relativa ao número de restrições m não contempla as restrições do tipo limite simples. O limite máximo de iterações é 1000, pelo que na coluna $\#it$ este

valor indica que o processo iterativo não convergiu para o critério de paragem estipulado em (5.5).

Os valores utilizados nas constantes existentes no algoritmo foram:

- $\delta = 1.0e^{-6}$, em (5.2) e (5.3) (usada na condição de aceitação do ponto intermédio z^k, e na condição de aceitação de x^{k+1});

- $\omega = 1.0e^{-20}$, em (5.4), (representa a tolerância de $h(x)$ no teste da inserção do par temporário no filtro);

- $\gamma = 1.0e^{-5}$ e $\beta = 1 - \gamma$, em (5.1) em que $0 < \gamma < \beta < 1$ (usadas no envelope do filtro);

- $TolAlfa = 1.0e^{-6}$ em (4.2.1) (constante relativa ao limite do valor de α no algoritmo de procura unidimensional);

- $\epsilon = \xi = 1.0e^{-6}$ definem as tolerâncias no critério de paragem (5.5).

Na análise dos resultados computacionais salientam-se algumas particulares identificadas em alguns problemas, apresentadas na coluna *obs* da tabela. Assim, os problemas identificados com *a*) representam os problemas que sofrem do efeito de Maratos, isto é, o algoritmo ao resolver estes problemas chega perto da solução em poucas iterações, mas no entanto, não converge, *i.e.*, não verifica o critério de paragem estipulado. Também foi identificado um problema que falhou porque apesar de convergir para um ponto estacionário, viola as restrições. Este problema foi mencionado como passível de acontecer pelos autores no estudo teórico da convergência. Os problemas identificados com *b*) representam as situações em que o subproblema (LP), da fase da admissibilidade, não foi resolvido com sucesso pela subrotina do LSSOL, tendo terminado com $INFORM = 2$. Mesmo assim, seis destes problemas acabaram por convergir apenas com a fase da optimalidade. Foram identificados dois problemas em que o subproblema (QP), da fase da optimalidade, não foi resolvido pela subrotina do LSSOL, terminando com $INFORM = 2$.

Tabela 6.1: Resultados numéricos gerais do Algoritmo IR com Filtros

Problema	n	m	#it	#f	#g	#insfiltro	#elemfiltro	obs
alsotame	2	1	3	31	6	2	2	
biggsc4	4	7	2	4	3	1	1	
bqp1var	1	0	1	1	1	0	0	
camel6	2	0	8	9	9	0	0	
cantilvr	5	1	1000	19962	1000	997	13	b)
cb2	3	3	1000	19981	1000	521	40	b)
cb3	3	3	1000	19981	1000	668	9	b)
chaconn1	3	3	1000	19962	1000	572	7	b)
chaconn2	3	3	1000	19962	1000	667	6	b)
congigmz	3	5	16	243	16	12	11	
dembo7	16	20	1000	20013	1010	672	13	a)
dipigri	7	4	1000	19941	1000	726	6	a)
dixmaana	15	1	1	1	1	0	0	
dixmaanc	15	1	1	5	1	0	0	
dixmaane	15	1	1	5	1	0	0	
dixmaanf	15	1	1	5	1	0	0	
dixmaang	15	1	1	5	1	0	0	
dixmaanh	15	1	1	5	1	0	0	
dixmaani	15	1	1	5	1	0	0	
dixmaanj	15	1	1	6	1	0	0	
dixmaank	15	1	1	6	1	0	0	
dixmaanl	15	1	1	5	1	0	0	
engval1	2	1	15	15	15	0	0	
engval4	5	1	9	47	9	0	0	
explin	120	1	20	58	22	0	0	
expquad	120	1	1	1	1	0	0	
fletcher	4	4	1000	39980	1999	2	2	
gigomez2	3	3	1000	19981	1000	523	38	a)
gigomez3	3	3	1000	19981	1000	688	8	a)
goffin	51	50	3	41	3	1	1	
harkerp2	100	1	18	18	18	0	0	
hatfldb	4	0	11	128	13	4	2	
hatfldc	4	0	12	13	12	0	0	
hatfldh	4	7	8	256	9	2	2	
himmelp1	2	0	6	26	6	0	0	
himmelp2	2	1	6	7	6	0	0	
himmelp3	2	2	4	4	4	1	1	
himmelp4	2	3	3	22	3	1	1	
himmelp5	2	3	3	3	3	1	1	
himmelp6	2	4	4	5	4	1	1	
hs3mod	2	0	1	1	1	0	0	
hs21mod	7	2	8	10	8	0	0	
hs35mod	2	1	1	1	1	0	0	
hs44new	4	5	3	4	3	1	1	
hs100mod	7	4	1000	19951	1000	995	7	a)
hs268	5	5	1	1	1	0	0	
hubfit	2	1	1	1	1	0	0	
humps	2	1	1	1	1	0	0	
liarwhd	36	1	19	19	19	0	0	
liswet2	103	100	1	1	1	0	0	
liswet3	103	100	1	1	1	0	0	
liswet4	103	100	1	1	1	0	0	
liswet5	103	100	1	1	1	0	0	
liswet6	103	100	1	1	1	0	0	
liswet7	103	100	1	1	1	0	0	
liswet8	103	100	1	1	1	0	0	
liswet9	103	100	1	1	1	0	0	
liswet10	103	100	1	1	1	0	0	
liswet11	103	100	1	1	1	0	0	
liswet12	103	100	1	1	1	0	0	
logros	2	1	2	6	2	0	0	
lootsma	3	2	10	145	12	1	1	
lsqfit	2	1	1	1	1	0	0	
madsen	3	6	1000	19924	1000	992	17	a)
makela1	3	2	1000	1	1000	1000	1	a)
makela3	21	20	23	284	23	19	18	b)
makela4	21	40	2	21	2	1	1	b)
matrix2	6	2	14	38	14	12	12	
mifflin1	3	2	1000	17966	1000	499	499	b)
mifflin2	3	2	3	22	3	1	1	
minmaxrb	3	4	2	21	2	1	1	
nondquad	100	1	18	18	19	0	0	
oslbqp	8	0	1	21	2	0	0	
pfit1	3	1	1	1	1	0	0	
pfit1ls	3	1	1	1	1	0	0	
pfit2	3	1	1	1	1	0	0	
pfit2ls	3	1	1	1	1	0	0	
pfit3	3	1	1	1	1	0	0	
pfit3ls	3	1	1	1	1	0	0	
pfit4	3	1	1	1	1	0	0	
pfit4ls	3	1	1	1	1	0	0	
polak1	3	2	10	104	10	6	6	b)
polak3	12	10	1000	19905	1000	629	11	a)
powell20	10	10	1	2	2	1	1	
power	10	1	15	15	15	1	1	
pspdoc	4	0	5	27	5	0	0	
qrtquad	12	1	15	41	17	1	1	

Tabela 6.1 (continuação)

Problema	n	m	#it	#f	#g	#insfiltro	#elemfiltro	obs
qudlin	12	1	42	3	2	2	0	
rosenmmx	5	4	1000	17974	1000	499	499	b)
s365mod	7	4	1000	61	1000	4	2	b) e c)
simbqp	2	0	1	1	2	0	0	
simpllpa	2	2	2	20	2	2	2	
simpllpb	2	3	2	20	2	2	2	
sineval	2	1	13	25	13	0	0	
sisser	2	1	1	7	2	0	0	
snake	2	2	1	1	1	0	0	
stancim	3	3	1000	19981	1000	645	5	a)
tf12	3	100	1000	20000	1000	634	4	a)
vardim	10	1	25	25	26	0	0	
womflet	3	3	1	1	2	0	0	
zecevic2	2	2	1	1	2	0	0	
zecevic3	2	2	1000	2500	1000	254	254	$h \neq 0$
zecevic4	2	2	3	4	4	0	0	
zy2	3	1	5	8	6	0	0	
branin	2	0	5	26	7	0	0	
chi	2	0	5	8	6	0	0	
hs5	2	0	3	23	4	0	0	
hs15	2	2	3	21	4	0	0	b)
hs23	2	5	13	51	13	2	1	
hs35	3	1	1	1	2	0	0	
hs44	4	6	6	9	7	1	1	
hs64	3	1	1000	19819	1000	998	9	a)
kowalik	4	0	11	33	12	0	0	
levy3	2	0	4	10	5	0	0	
osborne1	5	0	17	78	18	0	0	
powell	4	0	16	19	17	0	0	
price	2	0	6	13	7	0	0	
s324	2	2	1000	19962	1000	750	7	a)
schwefel	5	0	9	10	10	0	0	
shekel	4	0	13	57	14	0	0	
tre	2	0	4	5	5	0	0	
weapon	100	12	11	12	12	0	0	
fir_convex	11	243	1000	20000	1000	498	7	a)
fir_exp	12	244	1	2	2	1	1	
fir_linear	11	243	1	2	2	1	1	
fir_socp	12	244	4	26	6	1	1	
fermat_socp_eps	5	3	1000	20000	1000	717	9	b)
steiner_nonconvex	33	17	1000	19728	1000	13	13	b)
steiner_socp_eps	33	17	1000	20000	1000	710	38	a)
steiner_socp_vareps	33	17	1000	11111	1000	679	36	a)
hs001	2	0	24	33	25	0	0	
hs002	2	0	1000	19887	1001	993	1	a)
hs003	2	0	1	1	2	0	0	
hs004	2	0	2	1	3	1	1	
hs005	2	0	4	25	5	0	0	
hs011	2	1	1000	16256	1000	1000	1000	
hs012	2	1	1000	17966	1000	499	499	
hs015	2	2	1	1	2	0	0	
hs016	2	2	4	5	5	0	0	
hs017	2	2	20	138	21	3	3	
hs018	2	2	15	15	3	3	3	
hs019	2	2	2	19	1	0	0	
hs020	2	3	5	6	6	1	1	
hs021	2	1	1	2	2	0	0	
hs022	2	2	9	126	10	7	6	
hs023	2	5	13	51	14	2	1	
hs024	2	2	3	4	4	2	2	
hs025	3	0	25	45	26	0	0	
hs029	3	1	1000	1	1001	999	1	c)
hs030	3	1	2	12	3	0	0	
hs031	3	1	7	8	8	1	1	
hs033	3	2	7	8	8	0	0	
hs034	3	2	1000	38742	1001	129	85	
hs035	3	1	1	1	2	0	0	
hs036	3	1	2	1	3	1	1	
hs037	3	1	1000	10919	1001	532	48	
hs038	4	0	23	27	24	0	0	
hs043	4	3	1000	16953	1001	997	997	
hs044	4	6	6	8	7	1	1	
hs045	5	0	3	2	4	1	1	
hs059	2	3	7	8	8	0	0	
hs064	3	1	1000	19810	1001	990	9	b)
hs066	3	2	1000	19796	1001	746	9	a)
hs072	4	3	1	1	2	0	0	
hs076	4	3	1	1	2	0	0	
hs083	5	3	4	11	5	1	1	

Tabela 6.1 (continuação)

Problema	n	m	#it	#f	#g	#insfiltro	#elemfiltro	obs
hs084	5	3	3	4	4	0	0	
hs086	5	6	3	4	4	0	0	
hs089	3	1	1000	19890	1000	554	43	a)
hs095	6	4	2	3	2	1	1	
hs096	6	4	4	45	6	3	3	
hs097	6	3	9	27	9	5	5	
hs098	6	3	3	5	3	2	2	
hs100	7	4	1000	19941	1000	995	7	a)
hs102	7	6	25	203	49	12	9	
hs103	7	6	17	129	33	8	8	
hs104	8	6	1000	29253	1000	17	15	
hs106	8	6	630	11220	929	28	17	
hs116	9	13	1000	38335	1965	11	10	
hs117	15	5	11	11	11	0	0	
hs118	15	17	1	2	2	0	0	
s215	2	1	3	21	4	2	2	b)
s218	2	1	14	14	14	13	13	
s221	2	1	19	19	20	0	0	
s222	2	1	61	90	82	9	8	
s223	2	2	6	10	6	2	2	
s224	2	2	2	79	4	1	1	
s225	2	5	8	26	9	1	1	b)
s226	2	2	2	3	3	1	1	
s227	2	2	7	101	7	6	6	
s229	2	0	19	28	20	0	0	
s230	2	2	1000	19963	1001	833	4	b)
s231	2	2	21	25	22	0	0	
s232	2	2	2	2	3	1	1	
s233	2	1	7	7	8	0	0	
s234	2	1	13	17	14	1	1	
s236	2	2	2	2	3	1	1	
s237	2	3	2	21	3	1	1	
s238	2	3	6	13	7	1	1	
s239	2	1	2	22	3	0	0	
s242	3	0	2	2	3	0	0	
s244	3	0	9	31	10	0	0	
s249	3	0	8	7	8	0	0	
s250	3	1	2	1	3	1	1	
s251	3	1	1000	10919	1001	532	48	
s253	3	1	3	3	4	0	0	
s257	4	0	10	11	10	0	0	
s259	4	0	9	16	9	0	0	
s268	5	5	1	1	2	0	0	
s270	5	1	12	14	13	2	2	
s277	4	4	1000	20000	1000	709	3	
s278	6	6	1000	20000	1000	649	5	
s279	8	8	1000	20000	1000	683	2	
s280	10	10	1000	20000	1000	645	5	
s307	2	0	10	48	11	1	1	
s315	2	3	1000	17985	1000	998	998	b)
s326	2	2	412	539	413	21	13	
s328	2	0	5	26	6	0	0	
s329	2	3	120	3452	239	86	84	
s331	2	1	24	44	25	1	1	
s332	2	1	1000	6643	1525	152	129	
s337	3	1	6	45	7	2	2	
s341	3	1	2	4	3	1	1	
s342	3	1	3	6	4	1	1	
s343	3	2	1	2	2	1	1	
s346	3	2	1	2	2	1	1	
s354	4	1	1000	19848	1000	571	6	
s357	4	35	6	8	7	0	0	
s358	5	0	781	9281	782	3	1	
s359	5	14	1	1	2 0	0		
s360	5	2	1000	2000	1000	73	73	
s361	5	6	2	2	3 0	0		
s366	7	14	1000	19875	1010	510	21	
s368	8	0	2	22	3	2	2	
s372	9	12	1000	19981	1000	550	8	a)

Estes foram denotados por $c)$ na tabela. Este resultado da rotina do LSSOL significa que a solução, aparentemente, não tem limites - no procedimento interno do LSSOL, o tamanho do passo apresenta um valor muito grande, perto de infinito.

Na resolução do problema $fletcher$, a fase da admissibilidade sofre do mesmo problema

referido anteriormente, o subproblema (LP) termina com $INFORM = 2$ na primeira iteração, e o subproblema (QP), da fase da optimalidade, termina, em todas as iterações, com $INFORM = 3$. Este resultado da rotina do LSSOL significa que não foi encontrado nenhum ponto admissível, isto é, não foi possível satisfazer as restrições dentro da tolerância de admissibilidade da própria rotina.

Salienta-se ainda que no universo dos problemas testados, 183 deles convergiram, ou seja, 77%.

Analisando o comportamento do filtro, para os problemas que convergiram, em termos do número de inserções efectuadas e do número de elementos no filtro final, elaboraram-se os gráficos das Figuras 6.1 e 6.2. Nesta avaliação, salienta-se que o número de inserções e de elementos no filtro final é muito pequeno, menos de cinco em mais de 90% dos problemas.

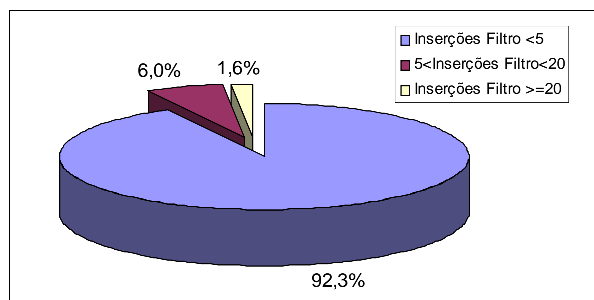

Figura 6.1: Inserções no filtro (problemas que convergem)

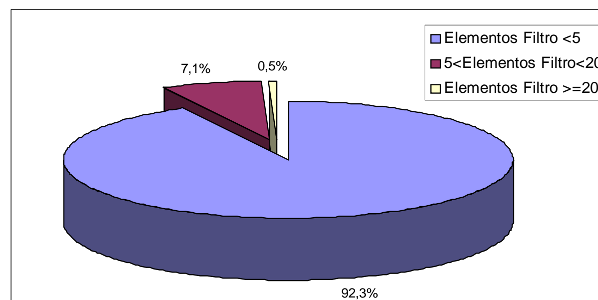

Figura 6.2: Elementos no filtro final (problemas que convergem)

A mesma análise, agora para os 52 problemas que não convergiram, permitiu constatar que o número de inserções no filtro aumenta consideravelmente (Figura 6.3). Verifica-se que em cerca de 60% dos problemas o número de inserções situa-se entre 400 e 800, e que em 25% é superior a 800. No entanto, no filtro final o número de elementos é consideravelmente mais baixo - inferior a 40 para 73% dos problemas (Figura 6.4).

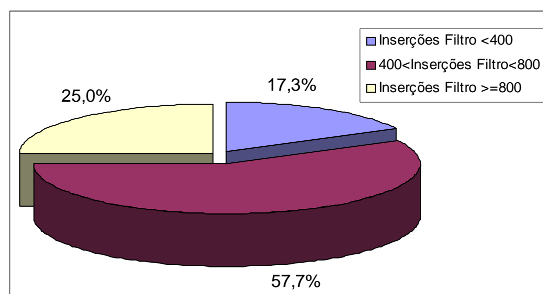

Figura 6.3: Inserções no filtro (problemas que não convergem)

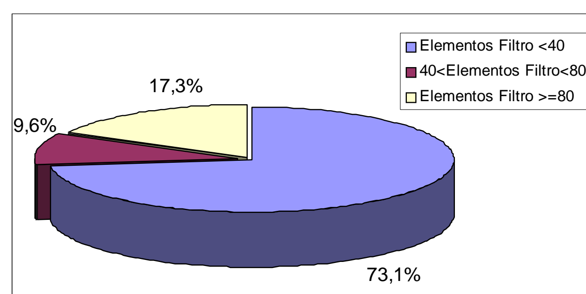

Figura 6.4: Elementos no filtro final (problemas que não convergem)

6.1.1 Avaliação da importância da fase da admissibilidade

No sentido de avaliar a importância da fase de admissibilidade no Algoritmo IR com Filtros, compararam-se duas versões do algoritmo - com as duas fases (admissibilidade e optimalidade) e com apenas a fase de optimalidade. O universo de problemas sobre o qual recaiu este teste é o conjunto de problemas em que houve convergência na Tabela 6.1. Assim, para este conjunto de problemas, houve necessidade, para aqueles em que o ponto inicial era admissível, testá-los com pontos não admissíveis, de forma a poder concluir acerca da importância da admissibilidade. Estes resultados estão na Tabela 6.2, em que as 3 primeiras colunas com valores numéricos se referem ao algoritmo com as duas fases e as 3 colunas seguintes à versão apenas com optimalidade. A coluna MS designa se o algoritmo nas duas versões convergiu para a mesma solução (S - Sim, N - Não).

Da análise desta tabela, salientam-se os problemas, identificados com e), que sem a fase da admissibilidade necessitam de menos iterações para convergirem, no entanto, convergem para pontos piores, isto é, têm pior valor de f. Também aconteceram quatro situações, identificadas por f), em que o algoritmo sem admissibilidade atingiu a mesma solução em menos iterações do que o algoritmo com admissibilidade.

Tabela 6.2: Análise comparativa do Algoritmo IR com Filtros com e sem fase de admissibilidade

	Algoritmo com Admissibilidade			Algoritmo sem Admissibilidade				
Problema	#it	#f	f	#it	#f	f	MS	obs
alsotame	3	31	0.082085	1000	19962	9,807872	N	
biggsc4	2	4	-24,375000	3	3	-24,375000	S	
camel6	8	9	-0,215464	7	26	2,104250	N	e)
explin	20	58	27,62383	16	26	39,29659	N	e)
hatfldb	11	128	0.005573	22	59	0.005573	S	
hatfldc	12	13	0	1000	0	14.106,1962	N	
hatfldh	8	256	-24,37500	1000	4493	-24,50000	N	
himmelp1	6	26	-62,053869	8	85	81,177341	N	
himmelp2	6	7	-62,053869	8	46	-8,198032	N	
himmelp4	3	22	-59,013124	3	2	-59,013124	S	
himmelp5	3	3	-59,013124	8	7	-59,013124	S	
himmelp6	4	5	-59,013124	38	206	-8,198032	N	
hs44new	3	4	-13	6	8	-15	N	
lootsma	10	145	1,414214	1000	19753	8,000000	N	
oslbqp	1	21	6,250000	1	1	6,250000	S	
powell20	1	2	57,812500	1000	20000	57,812500	S	
qrtquad	15	41	0,000071	1000	19830	0	N	
zecevic2	1	1	-4,125000	1	1	-3,125000	S	
zecevic3	629		97,306951	1000	19887	97,309450	N	
zecevic4	3	4	7,557508	4	4	7,557508	S	
zy2	5	8	2	7	25	2	S	
branin	5	26	0.397887	5	26	0.397888	S	
chi	5	8	-19,248788	5	43	279,324672	N	
hs5	3	23	1,228370	4	44	306,500000	N	
hs23	13	51	2,000000	1000	19910	9,472136	N	
hs35	1	1	0.111111	1	1	0.111112	S	
hs44	6	9	-15	7	12	-15	S	
kowalik	11	33	0.000307	11	13	0.000308	S	
levy3	4	10	-23,243044	2	1	-11,178666	N	e)
osborne1	17	78	0.024518	3	11	1,106036	N	e)
powell	16	19	0	19	19	0	N	
price	6	13	0	50	73	0	N	
schwefel	9	10	0	9	9	0	S	
shekel	13	57	-2,630472	8	19	-2,682860	N	
tre	4	5	0	7	8	0	N	
weapon	11	12	-1735,569580	9	9	-1735,569580	S	
fir_exp	1	2	1374,301869	1000	19981	1,046488	N	
fir_linear	1	2	0.045342	1000	20000	0.045342	N	
fir_socp	4	26	10,375	1000	20000	1,046488	N	
hs005	4	25	1,228370	3	22	1,228370	S	f)
hs015	1	1	306,500000	4	41	306,500000	S	
hs016	4	5	23,144661	5	24	23,144661	S	
hs017	10	51	1,000000	12	69	1,000000	S	
hs018	15	15	5	1000	19962	5	S	
hs020	5	6	40,198730	7	50	40,198730	S	
hs021	1	2	-99,960000	1	1	-99,960000	S	
hs023	13	51	2,000000	1000	19846	9,472136	N	
hs024	3	4	-1	2	21	0	N	e)
hs025	25	45	0	25	44	0	S	
hs030	2	12	1	2	2	1	S	
hs031	7	8	6,000001	6	45	6,000000	S	f)
hs033	7	8	-4,585786	1000	19772	2,000000	N	
hs038	23	27	0	19	19	0	S	f)
hs059	7	8	-6,749505	1000	12862	-7,840675	N	
hs086	3	4	-32,348679	3	3	-32,348679	S	
hs095	2	3	9,744780	1000	20000	0.015620	N	
hs096	4	45	0.461579	1000	19926	0.015620	N	
hs097	9	27	4,616290	1000	19981	3,135809	N	
hs098	3	5	9,081180	1000	19981	3,135809	N	
hs106	630	11220	20589,165670	1000	19887	6745,848611	N	
hs118	1	2	664,820450	1	1	664,820450	S	
s222	61	90	-1,500000	1000	17983	-1,662003	N	
s223	6	10	0	1000	17983	-1,662003	N	
s224	2	79	-304	2	39	-304	S	
s226	2	3	0	1000	17925	-3,808642	N	
s229	19	28	0	21	28	0	S	
s234	13	17	-0,8	23	30	0	N	
s236	2	2	-58,903436	88	531	-8,197517	N	
s237	2	21	-58,903436	9	8	-58,903436	S	
s238	6	13	-58,903436	1000	12936	-8,221491	N	
s239	2	22	-58,903436	88	531	-8,197517	N	
s244	9	31	0	10	14	0	S	
s270	12	14	-1	9	9	0	S	e)
s326	412	539	-79,807823	1000	13962	-81,122890	N	
s328	5	26	1,744152	11	49	1,744152	S	
s331	24	44	4,258385	36	196	4,258385	N	
s341	2	4	0,000000	1000	14972	-22,964810	N	
s342	3	6	0,000000	1000	140	0,000000	N	
s343	1	2	-0.000000	5	5	-5,684783	N	
s346	1	2	-0.000000	5	5	-5,684783	N	
s357	6	8	0.358457	6	7	0.358457	S	
s358	781	9281	0.000055	21	79	0.000055	S	f)
s368	2	22	0	2	21	0	S	

Para comparação destas duas versões do algoritmo, fez-se a análise comparativa de desempenhos, utilizando o gráfico de perfis de Dolan e Moré [10]. A interpretação deste gráfico não é fácil, sendo aconselhada a consulta do trabalho para esclarecimento de eventuais dúvidas. A medida a ser avaliada é o número de iterações. A análise da Figura 6.5 é ilucidativa: o gráfico da versão com admissibilidade tende mais rapidamente para 1, evidenciando a vantagem desta fase no algoritmo. Para a abcissa 0 a versão com admissibilidade apresenta um valor entre 0.8 e 0.9 - este facto significa que em mais de 80% dos problemas a versão com admissibilidade apresentou melhor desempenho.

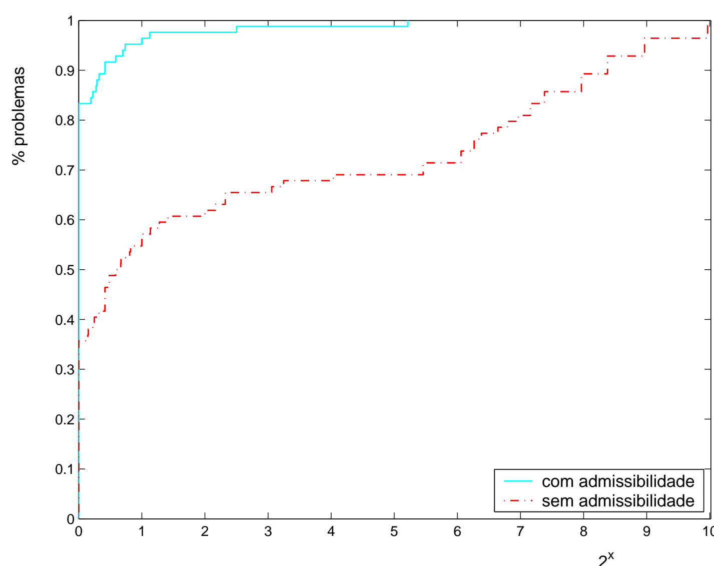

Figura 6.5: Perfis de desempenho com e sem admissibilidade

6.1.2 Comparação com o LOQO e NPSOL

O LOQO [43], criado por Robert Vanderbei, trata-se de um programa para resolver problemas de optimização suave com restrições. Para cada variável livre uma nova restrição é adicionada, implicitamente, fazendo com que as variáveis livres sejam escritas como a diferença entre dois números não negativos. As restrições de desigualdade são transformadas em igualdade, através da inclusão de variáveis de folga.

O LOQO é baseado num método primal-dual de pontos interiores aplicando uma sequência de aproximações quadráticas ao problema. Na fase primal, a admissibilidade

não é requerida no início do processo, apenas vai ser forçada ao longo dele. Os pontos interiores referem-se ao facto das variáveis de folga serem estritamente positivas durante todo o processo. Neste *solver*, a matriz Hessiana é exacta e, para induzir a convergência global usa a técnica de procura unidimensional.

NPSOL foi criado por Gill *et al.* [21]. É um *package* em *Fortran 77*, desenhado para resolver problemas de optimização não linear, onde as funções devem ser suaves mas não necessariamente convexas. Este algoritmo SQP apenas necessita das primeiras derivadas - quando estas não são fornecidas podem ser estimadas por diferenças finitas. A matriz Hessiana da função Lagrangeana é actualizada com a aproximação quasi-Newton BFGS.

O NPSOL utiliza um método baseado num esquema de procura unidimensional e envolve dois processos iterativos - um interno e outro externo. No processo iterativo interno resolve-se um subproblema quadrático para determinar a direcção de procura. Uma vez encontrada, o processo iterativo externo calcula uma nova aproximação à solução exigindo um decréscimo na função mérito (Lagrangeana aumentada). A sequência de aproximações geradas pelo processo iterativo externo converge para um ponto que verifica as condições de primeira ordem ou KKT. Como as restrições dos subproblemas são linearizadas e a função objectivo é uma aproximação quadrática à Lagrangeana, nem a função nem os valores do gradiente são necessários durante a solução de cada QP. Assim é especialmente eficaz para problemas não lineares cujas funções e gradientes envolvam extensos cálculos.

É de salientar que o subproblema (QP) interno é resolvido pela subrotina LSSOL. Este pacote interno resolve problemas lineares, quadráticos e de mínimos quadrados com restrições lineares. As restrições lineares e as de limites simples são tratadas separadamente por um método de conjunto activo. Possui um algoritmo numericamente estável e é especialmente recomendado para programação quadrática e programação linear densa. O LSSOL envolve duas fases para a resolução de um problema quadrático: na primeira, a fase de admissibilidade, é encontrado um ponto admissível através da minimização da violação das restrições; na segunda fase, a fase da optimalidade, a função objectivo quadrática é minimizada na região admissível.

Após se ter constatado que a versão com a fase de admissibilidade apresentou resultados

encorajadores, esta foi comparada com os pacotes de *software* LOQO e NPSOL. A Tabela 6.3 regista esses resultados. A coluna f indica o valor da função objectivo na solução encontrada.

A coluna *obs* da tabela identifica alguns pormenores identificados ao longo dos testes de comparação dos três *solvers*. Assim, o NPSOL, em alguns problemas, terminou com $INFORM = 6$, o que significa que a partir de determinada iteração não foi possível obter um decréscimo significativo da função mérito e por isso, o algoritmo ficou "estagnado" nesse ponto. Estes problemas estão identificados com a).

Nesta tabela comparativa nota-se que, no Algoritmo IR com filtros, a maioria dos problemas que não convergem, chegam à mesma solução, isto é, ao mesmo ponto que os dois outros *solvers*, na grande parte dos casos, ou pelo menos de um deles. Estes problemas estão identificados com b). Note-se, ainda, que grande parte destes problemas são os que foram identificados na Tabela 6.1 como sendo problemas que sofrem do efeito de Maratos.

Também foram identificados os problemas, com c), que obtiveram a mesma solução nos três *solvers* e que, nesses casos, o Algoritmo IR com Filtros atingiu a solução em menos de metade das iterações e numa grande parte destes casos a diferença do número de iterações ainda é maior do que o dobro.

Os resultados obtidos pelo algoritmo analisado aproximam-se mais dos obtidos pelo NPSOL, sendo em 34% dos casos obtida a mesma solução com um número de iterações inferior ou igual. Relativamente ao LOQO, o desempenho do Algoritmo IR com Filtros é significativamente melhor - o número de iterações é muito menor para o Algoritmo IR com Filtros, considerando os problemas que nos dois códigos convergiram para a mesma solução. O número de problemas nestas condições é de cerca de 41%.

Verificou-se ainda o caso de problemas em que o Algoritmo IR com Filtros obteve melhor solução, isto é, com menor valor de f, do que ambos os *solvers*, NPSOL e LOQO, ou pelo menos um deles, geralmente o LOQO. Este casos estão identificados por d).

Tabela 6.3: Análise comparativa do Algoritmo IR com Filtros com o LOQO e NPSOL

	Algoritmo IR com Filtros			LOQO				NPSOL				
Problema	#it	#f	f	#it	#f	f	MS	#it	#f	f	MS	obs
alsotame	3	31	0.082085	12	23	9,80787	N	4	6	9,807870	N	d)
biggsc4	2	4	-24,375000	18	35	-24.5	N	2	3	-24,375000	S	
bqp1var	1	1	0.000000	10	19	6,27E-04	S	1	2	0,000000	S	
camel6	8	9	-0,215464	12	23	-0,215464	N	8	15	-0,215464	S	
cantilvr	1000	19962	1,339956	16	31	1,33996	S	19	22	1,339960	S	b)
cb2	1000	19981	1,952224	11	21	1,95222	S	10	13	1,952220	S	b)
cb3	1000	19981	2	11	21	2	S	6	7	2	S	b)
chaconn1	1000	19962	1,952224	11	21	1,95222	S	8	11	1,952220	S	b)
chaconn2	1000	19962	2	11	21	2	S	5	6	2	S	b)
congigmz	16	243	28	33	65	28	S	4	5	28	S	
dembo7	1000	20013	174,786994	1000	2016	210,115	N	18	23	174,787000	S	
dipigri	1000	19941	680,630057	11	22	680.63	S	14	29	680,630000	S	b)
dixmaana	1	1	2975324,88574	79	278	1	N	24	33	1	N	
dixmaanc	1	5	168,114879	20	40	1	N	16	22	1	N	
dixmaane	1	5	101,354028	19	37	1	N	22	24	1	N	
dixmaanf	1	5	92,202303	20	40	1	N	22	24	1	N	
dixmaang	1	5	147,831452	23	52	1	N	23	25	1	N	
dixmaanh	1	5	266,591577	1000	2011	1,00017	N	27	31	1	N	
dixmaani	1	5	91,320176	19	39	1	N	39	42	1	N	
dixmaanj	1	6	77,878315	18	43	0,001725	N	29	52	-0,317278	N	a)
dixmaank	1	6	132,745476	18	38	0,001576	N	22	27	0	N	
dixmaanl	1	5	265,27504	18	41	0,000028	N	19	25	0	N	
engval1	15	15	0	20	39	0,001186	S	26	32	0	N	
engval4	9	47	3,632648	18	37	3,63265	S	19	34	3,632650	S	c)
explin	20	58	27,62383	103	315	0,000432	N	10	11	0,000001	N	
expquad	1	1	0	10	19	0,000033	N	2	3	0,000000	N	
fletcher	1000	39980	64	14	27	19,5254	N	1	1	4	N	a)
gigomez2	1000	19981	1,952224	11	21	1,95222	S	9	12	1,952220	S	b)
gigomez3	1000	19981	2	10	19	2	S	7	9	2	S	b)
goffin	3	41	0	12	23	0,001638	N	1	1	0,001638	N	d)
harkerp2	18	18	0.001583	38	75	0,000160	N	17	90	39622216,52	N	
hatfldb	11	128	0.005573	12	23	0,00557281	S	13	16	0,005573	S	
hatfldc	12	13	0	26	51	0,000000	S	22	30	0,000000	S	c)
hatfldh	8	256	-24,37500	20	39	-24.5	N	2	4	-24,375	S	
himmelp1	6	26	-62,053869	15	29	-62,0539	S	6	12	-23,8974	N	
himmelp2	6	7	-62,053869	19	38	-62,0539	S	30	31	-62,0539	S	c)
himmelp3	4	4	-59,013124	16	31	-59,0131	S	3	4	-59,0131	S	
himmelp4	3	22	-59,013124	16	32	-59,0131	S	10	12	-59,0131	S	c)
himmelp5	3	3	-59,013124	36	76	-59,0131	S	10	17	-59,0131	S	c)
himmelp6	4	5	-59,013124	25	51	-59,0131	S	9	11	-59,0131	S	c)
hs3mod	1	1	0	12	23	0,000308	S	4	8	0,000000	S	c)
hs21mod	8	10	-95,9600	30	60	-95,96	S	16	19	-95,96	S	c)
hs35mod	1	1	0,25	16	31	0,25	S	3	4	0,25	S	c)
hs44new	3	4	-13	18	35	-13	S	4	6	-15	N	
hs100mod	1000	19951	678,754727	15	29	678,754727	S	14	28	678,754727	S	b)
hs268	1	1	9994,418403	18	35	9994,42	S	15	22	9994,42	S	c)
hubfit	1	1	0,016893	12	23	0,0168935	S	6	7	0,016894	S	c)
humps	1	1	13669,381335	271	595	0,000522709	N	194	338	0,000000	N	
liarwhd	19	19	0,000001	27	55	0,000000	S	26	27	0,000000	S	
liswet2	1	1	-34,972170	19	37	-34,9722	S	2	3	-34,9722	S	c)
liswet3	1	1	-21,376876	19	37	-21,3769	S	2	3	-21,3769	S	c)
liswet4	1	1	-15,552873	23	45	-15,5529	S	2	3	-15,5529	S	c)
liswet5	1	1	-331,056756	16	31	-331,057	S	2	4	-331,057	S	c)
liswet6	1	1	-45,191411	19	37	-45,1914	S	2	3	-45,1914	S	c)
liswet7	1	1	-51,240974	28	55	-51,241	S	2	3	-51,241	S	c)
liswet8	1	1	-51,248168	21	41	-51,2482	S	2	3	-51,2482	S	c)
liswet9	1	1	-29,969457	77	153	-29,9695	S	2	3	-29,9695	S	c)
liswet10	1	1	-52,243763	19	37	-52,2438	S	2	3	-52,2438	S	c)
liswet11	1	1	-51,240981	52	103	-51,241	S	2	3	-51,241	S	c)
liswet12	1	1	-32,111149	75	149	-32,1111	S	2	3	-32,1111	S	c)
logros	2	6	0.668063	79	227	0	N	77	109	0,000019	N	
lootsma	10	145	1,414214	1000	11363	18,8825	N	1	1	0	N	a)
lsqfit	1	1	0,033787	12	23	0,033787	S	5	6	0,033787	S	c)
madsen	1000	19924	0,616432	24	48	0,616432	S	13	17	0,616432	S	b)
makela1	1000	1	0	14	28	-1,4142	N	9	13	-1,414210	N	
makela3	23	384	0	16	31	0,000563721	N	28	31	0,000000	N	
makela4	2	21	0	12	23	0,000000	S	20	56	0,000000	S	c)
matrix2	14	38	0	26	51	0,000934718	N	21	24	0,000000	N	
mifflin1	1000	17966	-1,087536	9	17	-1	N	8	10	-1	N	
mifflin2	3	22	-2,982063	13	25	-1	N	9	12	-1	N	d)
minmaxrb	2	21	0	13	25	0,000000	S	4	5	0	S	c)
nondquar	18	18	0	23	45	0,00198303	N	1000	1007	0,006102	N	
oslbqp	1	21	6,250000	40	80	6,25	S	1	2	6,25	S	
pfit1	1	1	372,119651	337	809	1,264	N	389	578	1,2682	N	
pfit1ls	1	1	372,119651	337	809	1,264	N	390	578	1,2682	N	
pfit2	1	1	3660,081151	339	802	35,9896	N	557	807	35,9741	N	
pfit2ls	1	1	3660,081151	339	802	35,9896	N	558	807	35,9741	N	
pfit3	1	1	15280,622347	345	840	253,874	N	433	612	253,973	N	
pfit3ls	1	1	15280,622347	345	840	253,874	N	434	612	253,973	N	
pfit4	1	1	43522,448838	334	816	943,326	N	503	736	943,297	N	
pfit4ls	1	1	43522,448838	334	816	943,326	N	504	736	944,297	N	
polak1	10	104	2,718282	14	27	2,71828	S	13	15	2,71828	S	
polak3	1000	19905	5,882365	24	47	5,8824	S	23	29	5,88237	S	b)
powell20	1	2	57,812500	12	23	57,8125	S	1	2	57,8125	S	
power	15	15	0	21	41	0,000984475	N	37	43	0	S	
pspdoc	5	27	2,414214	11	21	2,41421	S	12	13	2,41421	S	c)
qrtquad	15	41	0,000071	29	83	0,00261303	N	1	2	0	N	
qudlin	3	42	0.75	17	36	-0,000323227	N	4	5	0	N	
rosenmmx	1000	17974	-48,272044	15	29	-44	N	18	33	-44	N	
s365mod	1000	61	0,25	28	66	52,1399	N	26	39	52,1399	N	

Tabela 6.3 (continuação)

	IR com Filtros			LOQO				NPSOL				
Problema	#it	#f	f	#it	#f	f	MS	#it	#f	f	MS	obs
simbqp	1	1	0	13	25	0,000206861	S	4	6	0	S	c)
simpllpa	2	20	1	12	23	1	S	0	1	1	S	
simpllpb	2	20	1,1	13	25	1,1	S	2	4	1,1	S	
sineval	13	25	1,335260	48	107	0,0051374	N	81	119	0	N	
sisser	1	7	2,792658	19	38	0,00265704	N	40	42	0	N	
snake	1	1	0	1000	11367	-16,9088	N	272	564	0	N	
stancim	1000	19981	5	41	81	5,0000	S	2	11	7,46242	N	a) e b)
tf12	1000	20000	0,649031	15	29	0,649031	S	20	32	0,649031	S	b)
vardim	25	25	0	39	82	0,00185676	S	27	28	0	S	
womflet	1	1	0	11	21	6,05	N	182	519	0	N	
zecevic2	1	1	-4,125000	11	21	-4,125	S	2	4	-4,125	S	c)
zecevic3	1000	2500	97,308830	12	23	97,3095	S	8	11	97,3095	S	b)
zecevic4	3	4	7,557508	12	24	7,55751	S	6	7	7,55751	S	c)
zy2	5	8	2	14	27	2	S	6	16	2	S	
branin	5	26	0,397887	15	29	0,397887	S	7	10	0,397887	S	
chi	5	8	-19,248788	16	40	498,359	N	9	15	-34,2774	N	d)
hs5	3	23	1,228370	10	19	1,22837	S	7	9	1,228370	S	c)
hs15	3	21	306,5	16	31	306,5	S	5	6	306,5	S	c)
hs23	13	51	2,000000	12	23	9,47214	N	4	7	9,472136	N	d)
hs35	1	1	0,111111	11	21	0,111111	S	7	8	0,111111	S	c)
hs44	6	9	-15	12	23	-13	N	2	3	-13	N	d)
hs64	1000	19810	6299,842428	27	53	6299,84	S	29	39	6299,842428	S	b)
kowalik	11	33	0.000307	12	23	0,000307486	S	20	25	0,000307	S	
levy3	4	10	-23,243044	17	35	-28,479	N	5	9	-43,033178	N	
osborne1	17	78	0,024518	515	1444	0,0304108	N	148	214	0,000055	N	
powell	16	19	0	22	43	0,000052	S	59	67	0	N	
price	6	13	0	70	139	0,000000	N	25	32	0	N	
s324	1000	19962	5	15	29	4,99999995	S	13	23	5	S	b)
schwefel	9	10	0	9	17	0,000000	N	34	36	0,0000006	N	
shekel	13	57	-2,630472	19	37	-2,68286	N	24	57	-10,1531997	N	
tre	4	5	0	14	27	0,00000000	S	7	11	0	N	
weapon	11	12	-1735,569580	23	45	-1.735,56950	N	54	64	-1735,5696	N	
fir_convex	1000	20000	1,046488	40	79	1,0464876	N	4	6	1,0464876	S	b)
fir_exp	1	2	1374,301869	43	85	1,0464876	N	5	6	1,0464876	N	
fir_linear	1	2	0,045342	18	35	0,045342026	S	1	2	0,045342	S	
fir_socp	4	26	10,375	56	111	1,04648765	N	4	5	1,046488	N	
fermat_socp_eps	1000	20000	7,464102	11	21	7,46410163	S	9	10	7,464102	S	b)
steiner_nonconvex	1000	19728	0	1000	16718	25,3559837	N	53	200	0	N	
steiner_socp_eps	1000	20000	25,356285	27	53	25,3562848	S	55	107	25,3563	S	b)
steiner_socp_vareps	1000	20000	25,356068	64	359	25,3560678	N	82	190	25,3561	S	b)
hs001	24	33	0	32	63	0,00000000	S	17	20	0	S	
hs002	1000	19887	4,941229	21	41	4,94122932	S	9	14	0,050426	N	b)
hs003	1	1	0	11	21	0,00071575	S	3	8	0	S	c)
hs004	2	1	2,666667	8	15	2,66666668	S	1	2	2,66667	S	
hs005	4	25	1,228370	13	25	-1,91322296	N	7	13	1,228370	S	
hs011	1000	16256	-24,961059	13	25	-8,498460	N	8	12	-8,498464	N	
hs012	1000	17966	-34,775309	10	19	-30	S	8	9	-30	N	
hs015	1	1	306,500000	31	61	306,5	S	3	5	306,5	S	
hs016	4	5	23,144661	18	35	0,25	N	4	5	23,144661	S	
hs017	20	29	1,000002	29	58	1	S	12	15	1	S	
hs018	15	15	5	15	29	5	S	13	23	5	S	
hs019	2	17	-7972,993805	17	33	-6.961,81401	N	6	8	-6961,81388	N	d)
hs020	5	6	40,198730	16	31	40,1987	S	4	5	40,198730	S	
hs021	1	12	-99,960000	12	23	-99,9599999	S	2	4	-99,96	S	c)
hs022	9	9	1	9	17	1	S	5	7	1	S	
hs023	13	51	2,000000	13	26	9,47214	N	7	16	9,4721360	N	d)
hs024	3	4	-1	23	45	-1	S	1	4	-1	S	
hs025	25	45	0	15	30	0,00000000	S	0	1	32,835	S	
hs029	1000	10	-3,973389	10	19	-22,6274	N	12	15	-22,62742	N	
hs030	2	9	1	9	17	1	S	2	4	1	S	
hs031	7	17	6,000001	17	33	6	S	8	12	6	S	
hs033	7	8	-4,585786	20	39	2	N	7	8	-4,585786	S	d)
hs034	1000	14	-2,003258	14	27	-0,834032	N	7	8	-0,834032	N	
hs035	1	10	0,111111	10	19	0,111111119	S	5	7	0,111111	S	c)
hs036	2	16	-3300,000000	16	35	-3300	S	1	2	-3300	S	
hs037	1000	11	-3455,999251	11	21	-3455,99999	S	6	9	-3456	S	b)
hs038	23	27	0	25	50	0,00000000	S	26	35	0	S	
hs043	1000	11	-48,126124	11	21	-44	N	11	17	-44	N	
hs044	6	16	-15	16	31	-15	S	4	6	-15	S	
hs045	3	23	1	23	45	1	S	0	1	2	N	
hs059	7	22	-6,749505	22	47	-7,80279	N	13	17	-6,749505	S	
hs064	1000	27	6299,842428	27	53	6299,84	S	29	39	6299,84243	S	b)
hs066	1000	15	0.518163	15	29	0,518163	S	7	8	0,518163	S	b)
hs072	1	23	-4,681818	23	45	727,679	N	28	34	727,67936	N	d)
hs076	1	11	-4,681818	11	21	-4,68182	S	7	8	-4,6818182	S	c)
hs083	4	13	-30665,538672	13	25	-30665,5	N	5	7	-30665,53867	S	
hs084	3	4	-5280335,133215	20	39	-5280340	S	2	3	-5280335,133	S	
hs086	3	4	-32,348679	15	29	-32,3487	S	5	7	-32,348679	S	
hs089	1000	28	1,362657	28	58	1,36266	N	48	89	1,3626568	S	b)
hs095	2	16	9,744780	16	32	0,0156195	N	1	2	0,0156195	N	
hs096	4	45	0.461579	47	98	0,0156195	N	1	2	0,0156195	N	
hs097	9	18	4,616290	18	36	4,071250	N	3	6	3,135809	N	

Tabela 6.3 (continuação)

	IR com Filtros			LOQO				NPSOL				
Problema	#it	#f	f	#it	#f	f	MS	#it	#f	f	MS	obs
hs098	3	45	9,081180	45	137	3,13581	N	3	6	3,135809	N	
hs100	1000	11	680,630057	11	22	680,63	S	14	29	680,63006	S	b)
hs102	25	55	2736,708447	55	208	911,881	N	119	445	911,880571	N	
hs103	17	129	2185,868560	48	136	543,668	N	75	285	543,66796	N	
hs104	1000	14	3,950591	14	27	3,95116	S	18	20	3,951163	N	b)
hs106	630	11220	20589,165670	46	97	7049,25	N	17	21	7049,2480	N	
hs116	1000	38335	50	74	194	97,5875	N	10	20	97,5910345	N	
hs117	11	11	32,348680	34	67	32,3487	N	56	65	32,348679	N	
hs118	1	2	664,820450	22	43	664,82	S	14	28	664,82045	S	c)
s215	3	25	0	25	49	0,000000	S	6	8	0,0000	N	
s218	14	14	0	11	21	2,38457937	N	26	29	0	S	d)
s221	19	261	-0.999643	261	531	-1,00226	N	28	31	-1,000409	S	
s222	61	10	-1,500000	10	19	-1.5	S	4	6	-1,5	S	
s223	6	12	0	12	23	-0.834032	N	8	9	-0,834032	N	
s224	2	79	-304	12	23	-304	S	2	3	-304	S	
s225	8	17	2	17	33	2	S	6	7	2	S	
s226	2	9	0	9	17	-0,5	N	8	10	-0,5	N	
s227	7	101	1	9	17	1	S	6	8	1	S	
s229	19	28	0	27	56	0,00000000	S	32	41	0	S	
s230	1000	9	0,375	9	17	0.375	S	5	7	0,375	S	b)
s231	21	32	0	32	63	0,00000000	S	14	19	0	S	
s232	2	12	-1	12	23	-1	S	2	5	-1	S	
s233	7	13	0	13	25	0,00000000	S	26	47	2,11506	N	
s234	13	18	-0,8	18	35	-0,8	S	0	1	-0,8	S	
s236	2	17	-58,903436	17	33	-58,9034	S	7	10	-58,90344	S	c)
s237	2	46	-58,903436	46	99	-57,9034	S	13	22	-58,90344	S	c)
s238	6	1000	-58,903436	1000	11635	-818,773	N	8	11	-58,90344	S	
s239	2	15	-58,903436	15	30	-58,9034	S	16	24	-8,1975167	N	
s242	2	25	0	25	51	0,00012588	N	10	13	0	N	
s244	9	31	0	22	43	0,00213832	S	11	22	0	S	
s249	8	10	0	10	19	1	N	19	20	1	N	
s250	2	16	-3300	16	35	-3300	S	1	2	-3300	S	
s251	1000	11	-3455,999251	11	21	-3456	N	6	9	-3456	S	b)
s253	3	15	69,282032	15	29	69,282	S	7	8	69,28203	S	c)
s257	10	11	0,000000	26	51	0,00000000	S	25	36	0	S	c)
s259	9	16	-8,544621	12	23	-8,54462	S	21	37	3,887746	N	
s268	1	27	0	27	53	0,00000001	S	11	15	0	S	c)
s270	12	14	-1	17	33	0,00000017	N	5	8	-1	S	d)
s277	1000	13	5,076190	13	25	5,07619	S	5	12	5,076190	S	b)
s278	1000	13	7,838528	13	25	7,83853	S	10	26	7,8384893	N	a)
s279	1000	14	10,605950	14	27	10,6059	S	14	30	10,605950	S	a) e b)
s280	1000	16	13,375428	16	31	13,3754	N	12	25	13,37543	N	a)
s307	10	14	124,362182	14	27	124,362	S	15	28	124,36218	S	b)
s315	1000	14	-0.218011	14	27	-0,8	N	18	22	-0,8	N	
s326	412	12	-79,807823	12	23	-79,8078	S	6	9	-79,80782	S	
s328	5	22	1,744152	22	43	1,74415	S	10	15	1,7441520	S	c)
s329	120	3452	-6961,813878	18	37	-6961,81	S	8	11	-6961,8139	S	
s331	24	44	4,258385	9	20	4,25838	S	E				
s332	1000	6643	34,810161	1000	2152	29,3554	N	15	105	29,37979	N	
s337	6	11	6	11	21	6	S	7	10	6	S	
s341	2	11	0,000000	11	21	-22,62740	N	12	15	-22,627417	N	
s342	3	23	0,000000	23	45	-22,62740	N	112	213	-22,627417	N	
s343	1	27	-0.000000	27	53	-5,68478	N	5	9	-5,684782	N	
s346	1	27	-0.000000	27	53	-5,68478	N	5	9	-5,684782	N	
s354	1000	16	0,113784	16	32	0,113784	S	17	25	0,1137838	S	b)
s357	6	8	0,358457	21	42	0,358457	S	21	22	0,3584571	S	c)
s358	781	9281	0,000055	141	281	73783200	N	42	62	0,000055	S	d)
s359	1	17	-5504450,898864	17	33	-5504450	S	1	2	-5504450,899	S	
s360	1000	28	-5273583,389236	28	92	-5280340	S	2	3	-5280335,133	S	b)
s361	2	27	-15260,159095	27	66	-15260,2	S	3	4	-15260,1591	S	
s366	1000	47	1226,972633	47	95	1226,97	S	9	11	1226,97263	S	b)
s368	2	22	0	20	41	-0,9375	N	1	1	0	N	
s372	1000	32	13390,093119	32	65	13390,1	S	336	942	23273,2948	N	b)

Efectuou-se também a análise comparativa dos três códigos, em termos do número de iterações, utilizando os perfis de Dolan e Moré. Este facto é registado na Figura 6.6 - para a abcissa 0 o perfil do Algoritmo IR com filtros apresenta o valor de 0.60, significando que para 60% dos problemas o desempenho deste algoritmo é melhor do que qualquer um dos outros dois pacotes. Todavia, existe ainda um conjunto de problemas que prejudicam esta análise de desempenho. De facto, os problemas que sofrem do efeito de Maratos, isto é,

não verificam o critério de paragem mas já se encontram muito próximo da solução do problema, penalizam esta análise. Esta penalização, em termos de perfis, manifesta-se por uma tendência tardia para 1 do perfil correspondente ao Algoritmo IR com filtros.

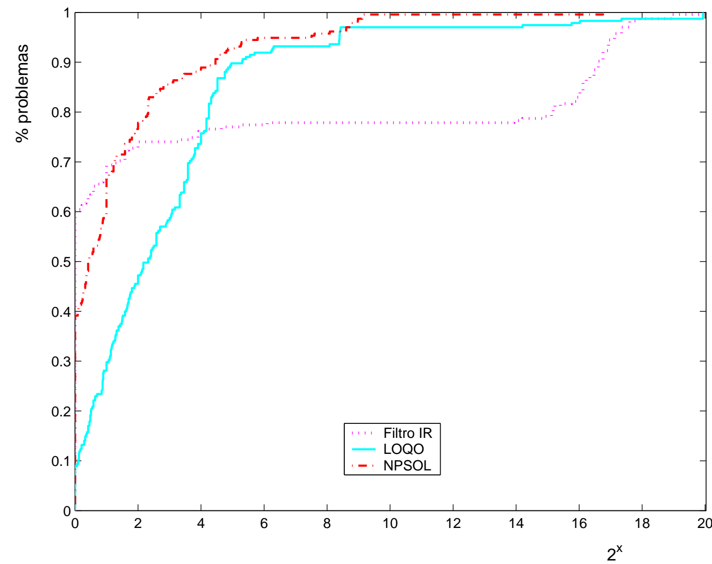

Figura 6.6: Perfis de desempenho dos três códigos

Capítulo 7

Conclusões e Trabalho Futuro

Este capítulo apresenta algumas conclusões, resultado de uma reflexão sobre o trabalho desenvolvido. Estando este trabalho numa fase quase embrionária, são definidas as linhas que orientarão a continuidade deste projecto num futuro próximo.

O algoritmo implementado encontra-se em fase de desenvolvimento e muitas melhorias podem ser introduzidas. Havendo ainda muito trabalho a desenvolver, algumas conclusões podem já ser inferidas, sem que sejam interpretados como prematuras. Assim, em termos da filosofia IR, constatou-se que a existência de uma fase de admissibilidade, cujo objectivo era encaminhar o processo iterativo para a região admissível acelerou consideravelmente o algoritmo. A comparação do algoritmo com e sem as duas fases da abordagem IR, permitiu avaliar a importância da fase da admissibilidade.

O método dos filtros implementado apresenta-se como uma estratégia simples e eficiente para substituir as funções de penalidade, evitando os problemas inerentes a estas. A existência da entidade "par temporário" originou filtros de menor dimensão, permitindo uma gestão mais fácil. A utilização deste método com procura unidimensional, evita a resolução de um novo subproblema quando o ponto candidato não é aceite pelo filtro. De facto, o valor do passo de procura é diminuído e o novo ponto é novamente testado - nas regiões de confiança, a rejeição de um ponto pelo filtro obriga a uma nova resolução de um novo subproblema para cálculo do candidato a próximo ponto do processo iterativo.

Em relação a sugestões possíveis para trabalho futuro podem referir-se as que se seguem:

- extender o desenvolvimento do algoritmo para o tratamento das restrições do tipo igualdade;
- avaliar o algoritmo com outras medidas de desempenho, tais como o tempo;
- estudar e implementar diferentes esquemas para estimação do vector dos multiplicadores de Lagrange;
- introduzir correcções de segunda ordem para evitar o efeito de Maratos;
- analisar as eventuais dificuldades provenientes da rotina LSSOL;
- resolver problemas de maior dimensão;
- implementar o procedimento de restauração do filtro, quando o valor de α na procura unidimensional atinge a tolerância estipulada;
- analisar a convergência do algoritmo final, já contemplados os algoritmos internos;
- estudar com mais detalhe o funcionamento e gestão do filtro;
- desenvolver uma interface gráfica de suporte ao algoritmo.

Bibliografia

[1] António Sanches Antunes and M. Teresa T. Monteiro. A SQP-filter algorithm with line search in nonlinear programming. XXVIII Congreso Nacional de Estatística e Investigación Operativa, Cádiz, 25-29 de Outubro, 2004, (9 páginas) CDROM, ISBN 84-689-0438-4.

[2] Charles Audet and J. E. Dennis Jr. A pattern search filter method for nonlinear programming without derivatives. *SIAM Journal on Optimization*, 14(4):980–1010, 2004.

[3] Mokhtar S. Bazaraa, Hanif D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. John Wiley & Sons, Inc., 1993.

[4] H. Y. Benson, D. F. Shanno, and R. J. Vanderbei. Interior-point methods for nonconvex nonlinear programming: Filter methods and merit functions. *Computational Optimization and Applications*, 23:257–272, 2002.

[5] N.; Martínez J. M Birgin, E. G.; Krejic. Solution of bounded nonlinear systems of equations using homotopies with inexact restoration. *International Journal of Computer Mathematics*, 80(2):211–223, 2003.

[6] Choon Ming Chin and Roger Fletcher. Numerical performance of an SLP - filter algorithm that takes EQP steps. Technical Report NA/202, University of Dundee, 2001.

[7] Choong Ming Chin. Numerical results of SLPSQP, filterSQP and LANCELOT on selected CUTE test problems. Technical Report NA/203, University of Dundee, 2001.

[8] Choong Ming Chin. A global convergence theory of a filter line search method for nonlinear programming. Technical report, Department of Statistics, University of Oxford, 1 South Parks Road, Oxford OX1 3TG, England, UK, 2002.

[9] Choong Ming Chin and Roger Fletcher. On the global convergence of and SLP-filter algorithm that takes EQP steps. Technical Report NA/199, Department of Mathematics, University of Dundee, Scotland, 2001. To appear in Mathematical Programming.

[10] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematics Programming*, A 91:201–213, 2001.

[11] Suzana L. C. Castro e Ana Friedlander. Algoritmo de restauração inexacta aplicado à resolução de programação matemática em dois níveis. Technical Report 01, Instituto de Matemática, Estatística e Computação Científica, Universidade Estadual de Campinas, Julho 2004.

[12] Roger Fletcher and Sven Leyffer. A bundle filter method for nonsmooth nonlinear optimization. Technical Report NA/195, Department of Mathematics, University of Dundee, Scotland, 1999.

[13] Roger Fletcher and Sven Leyffer. Nonlinear programming without a penalty function. *Mathematics Programming*, 91:239–269, 2002.

[14] Roger Fletcher, Sven Leyffer, and Philipe L. Toin. On the global convergence of an SLP-filter algorithm. Technical Report NA/183, Department of Mathematics, University of Dundee, Scotland, 1998.

[15] Roger Fletcher, Sven Leyffer, and Philippe L. Toint. On the global convergence of a filter-SQP algorithm. *SIAM Journal on Optimization*, 13:44–59, 2002.

[16] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modelling Language for Mathematical Programming*, 1993.

[17] Birgin E. G., Natasa Krejic, and J. M. Martínez. Solution of bounded nonlinear systems of equations using homotopies with inexact restoration. *International Journal of Computer Mathematics*, 80:211–222, 2003.

[18] Birgin E. G. and J. M. Martínez. Local convergence of an inexact-restoration method and numerical experiments. *Journal of Optimization Theory and Applications*, 127(2):229 – 247, 2005.

[19] David M. Gay. Hooking your solver to AMPL. Technical Report 97-4-06, Computing Sciences Research Center, Bell Laboratories, 1997.

[20] Philip E. Gill, Sven J. Hammarling, Walter Murray, Michael A. Saunders, and Margaret H. Wright. User's guide for LSSOL: A fortran package for constrained linear least-squares and convex quadratic programming. Technical Report 86-1, Systems Optimization Laboratory, Department of Operations Research, Stanford University, 1986.

[21] Philip E. Gill, Walter Murray, Michael A. Saunders, and Margaret H. Wright. User's guide for NPSOL: A fortran package for nonlinear programming. Technical Report 86-2, Systens Optimization Laboratory, Department of Operations Research, Stanford University, 1986.

[22] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press.

[23] Clóvis C. Gonzaga, Elizabeth Karas, and Márcia Vanti. A globally convergent filter method for nonlinear programming. *SIAM Journal on Optimization*, 14:646–669, 2003.

[24] N. I. M. Gould, C. Sainvitu, and Philippe L. Toint. A filter-trust-region method for unconstrained optimization. *SIAM Journal Optimization*, 16(2):341–357, 2005.

[25] N. I. M Gould and Ph. L. Toint. The filter idea and its application to the nonlinear feasibility problem. To appear in the Proceedings os the Dundee 2003 Conference on Numerical Analysis, 2003.

[26] N. I. M. Gould and Ph. L. Toint. FILTRANE, a fortran 95 filter-trust-region package for solving nonlinear feasibility problems. Technical Report 03/17, University of Namur, 2003.

[27] N. I. M. Gould and Philippe L. Toint. Global convergence of a hybrid trust-region SQP-filter algorithm for general nonlinear programming. *System Modeling and Optimization XX*, pages 23–54, 2003.

[28] Nicholas I. M. Gould, Sven Leyffer, and Philippe L. Toint. A multidimensional filter algorithm for nonlinear equations and nonlinear least-squares. *SIAM Journal Optimization*, 15:17–38, 2004.

[29] N. I. M. Goulg and Ph. L. Toint. How mature is nonlinear optimization? 2003.

[30] Elizabeth Karas, Ademir Ribeiro, Claudia Sagastizábal, and Mikhail Solodov. A bundle-filter method for nonsmooth convex constrained optimization. to appear, 2005.

[31] Martínez J. M. and Pilotta E. A. Inexact restoration methods for nonlinear programming: Advances and perspectives. *Optimization and Control with Applications*, pages 271–292, 2005.

[32] J. M. Martínez. A two-phase trust-region model algorithm with global convergence for nonlinear programming. *Journal of Optimization Theory and Applications*, 96:397–436, 1998.

[33] J. M. Martínez. Inexact-restoration method with lagrangian tangent decrease and new merit function for nonlinear programming. *Journal of Optimization Theory and Applications*, 111:39–58, 2001.

[34] J. M. Martínez and E. A. Pilotta. Inexact-restoration algorithm for constrained optimization. *Journal of Optimization Theory and Applications*, 104:135–163, 2000.

[35] José Mario Martínez. A semifeasible trust-region model algorithm for minimization with inequality constrints. *Novi Sad Journal of Mathematics*, 28:1–29, 1998.

[36] K. M. Miettinen. *Multiobjective Optimization.* International Series in Operations Research & Management Science. Kluwer Academic Publishers, 1999.

[37] Stephen G. Nash and Ariela Sofer. *Linear and Nonlinear Programming.* McGraw-Hill International Editions, 1996.

[38] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer Series in Operation Research, 1999.

[39] J. L. Chela e A. Friedlander R. Andreani. Resolução do problema de congestionamento urbano utilizando restauração inexacta. In *XXVIII - CNMAC: Congresso Nacional de Matemática Aplicada e Computacional.* SBMAC - Sociedade Brasileira de Matemática Aplicada e Computacional, 2005.

[40] M. Ulbrich and S. Ulbrich. Non-monotone trust region methods for nonlinear equality constrained optimization without a penalty function. *Mathematical Programming*, B(95):103–135, 2003.

[41] Michael Ulbrich, Stefan Ulbrich, and Luís N. Vicente. A globally convergent primal-dual interior-point filter method for nonlinear programming. *Mathematical Programming*, 100(2):379–410, 2004.

[42] Stefan Ulbrich. On the superlinear local convergence of a filter-SQP method. *Mathematical Programming Ser. B*, 100:217–245, 2004.

[43] R. J. Vanderbei. LOQO user's manual, version 4.05. Technical Report ORFE-99, Operations Research and Financial Engineering, Princeton University, October 1999.

[44] Andreas Wächter and Lorenz T. Biegler. Line search filter methods for nonlinear programming: Local convergence. *SIAM Journal Optimization*, Vol. 16(No. 1):32–48, 2005.

[45] Andreas Wächter and Lorenz T. Biegler. Line search methods for nonlinear programming: Motivation and global convergence. *SIAM Journal of Optimization*, 16(1):1–31, 2005.

[46] Andreas Wächter and Lorenz T. Biegler. On the implementation of a interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 100(1):25–57, 2006.

Apêndices

Apêndice A

Código Fonte do Algoritmo

filterphased.c

```
#include "filterphased.h"
#include <stdio.h>
#include <time.h>

struct tm inicio, fim;
//struture to count CPU time
//struct timeval tstart;



//AMPL
struct Options opt;

static double objsign; // maximization or minimization problem
static fint NERROR = -1;

#define asl cur_ASL

char filter_version[]="FILTERPHASED v1.0";

/* This struct member names must be in alphabetic order,
   for binary search */

keyword keywds[] = {
  KW("exemplod"   , filter_opt_d, (Char*)&opt.exemplod,
     "Exemplo de uma op\c{c}{\~a}o com double"),
  KW("iexemplo"   , filter_opt_i, (Char*)&opt.exemploi,
     "Exemplo de uma op\c{c}{\~a}o com um inteiro")
};


/*Definition of solver options:
  - "filterphased" -> invocation name of the solver
  - "FILTERPHASED" -> solver name in startup banner
  - "filter_options" -> name of solver_options environment var
  - keywds -> key words
  - nkeywds -> number of key words -->Onde {\'e} que isto {\'e} definido?????
  - 1 -> "want_funcadd = 1", lists the available user-defined functions
  - filter_version -> for -v (reports the solver version)
  - 0
  - NULL
*/

struct Option_Info Oinfo = { "filterphased", "FILTERPHASED",
"filter_options",
                 keywds, nkeywds, 1, filter_version, 0, NULL};




/************************************************ Set options.
Integer type ************************************************/ char
*filter_opt_i(Option_Info *oi, keyword *kw, char *value) {
  long optval;
  char *s;

  /* never echo options */
  oi->option_echo &= ~ASL_OI_echo;
```

```
  optval=strtol(value, &s, 10);
  if(s > value){
    /* existing integer number */
    *(int *)kw->info=(int)optval;
    printf("\nDefault option %s=%d changed\n", kw->name, *(int *)kw->info);
    return s;
  }

return value; }


/********************************************** Set options.
Double type *********************************************/ char
*filter_opt_d(Option_Info *oi, keyword *kw, char *value) {
  double optval;
  char *s;

  /* never echo options */
  oi->option_echo &= ~ASL_OI_echo;

  optval=strtod(value, &s);

  if(s > value){
    /* existing double number */
    *(double *)kw->info=optval;
    printf("\nDefault option %s=%.6f changed\n", kw->name, *(double *)kw->info);
    return s;
  }

return value; }

/*Errors report*/

static Jmp_buf Jb;

void catchfpe(int n) {
  report_where(asl);
  printf("\nFloating point error.\n");
  fflush(stdout);
  longjmp(Jb.jb,1);
}

int main(int argc, char **argv) {
    List aux,filter, nodexk, nodezk, nodexk1, pointerxk, pointerzk, pointerxk1;
    double *xk, *zk, *xk1, *dfea, *dopt, *constx, *constz, *constx1, *lambda, *constauxz, *constaux;
    double *hessianLagr, *jacobian, *gradObj, *CLAMBDA, *CLAMBDAaux;
    double *evalc, *CVEC,*BL,*BU;
    double fxk=0, fzk=0, hxk=0, hzk=0, alfa=0 , miu=0, residuo=0;
    double fxk1=0, hxk1=0, faux=0, haux=0, fauxz=0, hauxz=0;
    int k, i, p, cp, converge, Accept, AcceptInFilter, cont=0, contalfa=0, numlc=0, contFiltro=0, contG=0;
    int contfea=0, contfeaF=0, contopt=0, contoptF=0, r=0, contfeaG=0;
    long INFORM=0;


    float telapsed;
    clock_t start, end;


    FILE *fich, *nl;

    /*Variables for AMPL*/

    char *stub; /*file of type ".nl"*/
    ASL *asl;   //structure that store all the components that will be read from the file
    fint m, n, no, nz, mxr, mxc;
    double *lb, *ub, *lbt, *ubt, *tmp;
    int nonlc;

    //starts counting CPU time
    //start_timer();
    start= time(&inicio);    // come\c{c}a a medir o tempo

    /*Pointers to number of variables (n_var), number of constraints (n_conv) and number of objective functions (n_obj),
      defined in asl.h
      ASL_read_pfgh -> sets that we want first and second derivates and partially separable structure*/
    asl = ASL_alloc(ASL_read_pfgh);

    /*"Stub".nl
      getstops calls functions getstub and getopts and returns a stub, complaining and exiting if none is found.
      getstub -> provides a small default set of command-line options
      getopts -> looks first in $solver_options and then at command-line for keywords and optional values*/
    stub = getstops(argv, &Oinfo);

    /*jac_dim_ASL  -> reads the first part of file nl (stub.nl) and records some numbers in the folowing variables:
      m = n_conv   -> number of constraints
      n = n_var    -> number of variables
      no = n_obj   -> number os objective functions
```

```
  nz = ncz     -> number of Jacobian nonzeros
  mxr = maxrow -> length of longest contraint name
  mxc = maxcol -> length of longest variable name*/
nl=jac_dim_ASL(asl, stub, &m, &n, &no, &nz, &mxr, &mxc, (fint) strlen(stub));
if (!nl){
    printf("Can't read problem\n");
    exit(1);
}
want_deriv = 1; // we want derivatives
want_xpi0 = 1; // we want initial primal guess

/*Reads the rest of nl file*/
pfgh_read_ASL(asl, nl, 0);

if(n_obj<1){
    printf("At least one objective is requested\n");
    exit(1);
}

if(n_obj>1){
    printf("Current implentation only supports one objective function\n");
    exit(1);
}

/* Use Dense matrix*/
dense_j();

/*Sets if OF is maximization or minimization*/
objsign = objtype[0] ? -1. : +1.;


fich = fopen("results.txt","w"); // opening a file (results.txt) to report the results

err_jmp1 = &Jb; // if err_jmp1 <> 0 then abort execution
if (!setjmp(Jb.jb)){
    signal(SIGFPE, catchfpe); //control of errors

    /* Fill upper and lower bounds arrays
       lb -> lower bounds
       ub -> upper bounds*/

    /* lower and upper bounds on variables and constraints */
    lbt = lb = (double *) getmemory((n+m)*sizeof(double));
    ubt = ub = (double *) getmemory((n+m)*sizeof(double));


    /*LUv -> array of alternating lower and upper variable bounds - n variables */
    tmp=LUv; // variables
    for(i=0;i<n;i++){
        *lbt++=*tmp++;
        *ubt++=*tmp++;
    }

    /*LUrhs -> array of alternating lower and upper constraints bounds - m constraints */
    tmp=LUrhs; // constraints
    for(i=0;i<m;i++){
        *lbt++=*tmp++;
        *ubt++=*tmp++;
    }

    /*Allocation of memory for points Xk, Zk and Xk+1*/
    xk = (double *) getmemory(n*sizeof(double));  //point Xk
    zk = (double *) getmemory(n*sizeof(double));  //point Zk
    xk1 = (double *) getmemory(n*sizeof(double)); //point Xk+1

    /*Allocation of memory for contraints value in point Xk, Zk and Xk+1*/
    constx = (double *) getmemory(m*sizeof(double));
    constz = (double *) getmemory(m*sizeof(double));
    constx1 = (double *) getmemory(m*sizeof(double));

    constauxz=(double *) getmemory(m*sizeof(double));
    constaux=(double *) getmemory(m*sizeof(double));

    /*Initialization of vectors*/
    initDoubleVector(xk,n);
    initDoubleVector(zk,n);
    initDoubleVector(xk1,n);
    initDoubleVector(constx,m);
    initDoubleVector(constz,m);
    initDoubleVector(constx1,m);

    initDoubleVector(constauxz,m);
    initDoubleVector(constaux,m);

    //Initializing Lagrange multipliers - m constraints; n variables
    lambda = (double *) getmemory((m)*sizeof(double));  //usar a fun\c{c}{\~a}o EGPP se n{\~a}o houver fase de optimalidade
    CLAMBDA = (double *) getmemory((n+m)*sizeof(double));
    CLAMBDAaux = (double *) getmemory((n+m)*sizeof(double));
```

```
for (i=0; i<n+m; i++)
    CLAMBDA[i] = 1.0;

for (i=0; i<n+m; i++)
    CLAMBDAaux[i] = 1.0;

for (i=0; i<m; i++)
    lambda[i] = 1.0;

/*Allocation of memory for dxk, dzk (direction), gradObj (gradient), hessianLagr (hessian) and jacobian*/
dfea = (double *) getmemory(n*sizeof(double));
dopt = (double *) getmemory(n*sizeof(double));
gradObj = (double *) getmemory(n*sizeof(double));
hessianLagr = (double *) getmemory((n*n)*sizeof(double));
jacobian = (double *) getmemory((n*m)*sizeof(double));

/*Initialization of vectors*/
initDoubleVector(dfea,n);
initDoubleVector(dopt,n);
initDoubleVector(gradObj,n);
initDoubleVector(hessianLagr,n*n);
initDoubleVector(jacobian,n*m);

//Inicializa\c{c}{\~a}ode vectores auxiliares
evalc = (double *) getmemory((n+m)*sizeof(double));  //vector que avalia as restri\c{c}{\~o}es violadas (0) ou n{\~a}o (1)

//Initialization of vector evalc
initDoubleVector(evalc,n+m);

if(X0) // AMPL provides initial guess -> X0
    memcpy(xk, X0, n*sizeof(double)); // Xk <- X0
else  // Zero initial guess
    memset(xk,0, n*sizeof(double));   // Xk <- 0

lagscale(1.0,NULL); //+ in Lagrangian function
//lagscale(-1.0,NULL); //- in Lagrangian function

/*  jacval   -> computes the Jacobian matrix of constraints evaluated by conval and stores it in jacobian
             cgrad is a structure like {varno, goff, coef}
             where goff values determine where in jacobian vector the nonzeros get stored*/
jacval(xk, jacobian, NULL);
objgrd(0, xk, gradObj, NULL);
contG=contG+1;

/*conval -> evaluates the bodies of contraints at point Xk and stores them in constx : LUrhs <= body <= Urhsx */
conval(xk, constx, NULL); // value of constraints

/*Computation of H(Xk)*/
hxk = Func_H(lb,ub,constx,xk,n,m);

/*objval -> returns the value of objective function nobj (number of OF) at the point X : nobj = 0 -> 1 O.F. */
fxk = objsign*objval(0,xk,NULL); // value of O.F.

filter = NewNode(-infinity, u, 0.); /* Inicialize the filter with upper
                                     bound on constraint infeasibility */

//Calculates the number of real contraints, i{\'e}, lb and ub != infinity
//Auxiliar function to solve the problem in solving LPs without constraints
//"r" is the number of real constraints
r=numConstraints(m,n,ub,lb);

//Number of linear constraints
numlc=m-nlc;

miu = 1.0;

nodexk = NewNode(fxk,hxk,miu); //initialization of filter
if ((nodexk->f <= filter->f) || (nodexk->h >= filter->h))
    printf("wrong choice of upper bound on constraint infeasibility.\n");


AcceptInFilter = FALSE;
converge = FALSE;
k = 1;
while ((!converge) && (k <= TolIter)) {  //Main cicle

    FuncEvalC(n,m,lb,ub,constx,xk,nonlc,evalc,&p,&cp);  //verifies wich constraints are violated

    contalfa=0;  //count of alpha divisions

    if (hxk <= zero) {           //CASE 1: all constraints are satisfied - no feasibility phase
        memcpy(zk, xk, n*sizeof(double));
        hzk = hxk;
        fzk = fxk;
        memcpy(constz,constx,m*sizeof(double)); // Value of constraints in point Zk
    }
    else {                      //CASE 2: there are violated constraints
```

```
    if (r==p && p==1) {     //there is no feasible phase!
        memcpy(zk, xk, n*sizeof(double));
        hzk = hxk;
        fzk = fxk;
        memcpy(constz,constx,m*sizeof(double)); // Value of constraints in point Zk
    }
    else {   //(p>1) OR (r>=p) - decision will be made inside of function solveLSSOLP)
        //*****************FEASIBILITY PHASE****************
        //calculates feasibility direction - LP problem

        solveLSSOLP(m,n,p,r,numlc,cp,lb,ub,evalc,xk,constx,jacobian,dfea,&INFORM,CVEC, BL, BU);

        if (INFORM==2) {
            memcpy(zk, xk, n*sizeof(double));
            hzk = hxk;
            fzk = fxk;
            memcpy(constz,constx,m*sizeof(double)); // Value of constraints in point Zk
        }
        else{
            contfea=contfea+1; //count of number of feasibilities cicles
            Accept = FALSE;
            alfa = 1.0;

            while ((!Accept) && (alfa >= TolAlfa)) { //Cycle to find direction on feasibility phase

                //Obtain Zk point
                for(i=0;i<n;i++)
                    zk[i]=xk[i]+alfa*dfea[i];

                fzk = objsign*objval(0,zk,NULL);   // Value of O.F in point Zk

                contfeaF=contfeaF+1;               //count of F.O. in feasibility phase

                conval(zk,constz,NULL);          // Value of constraints in point Zk

                hzk = Func_H(lb,ub,constz,zk,n,m); //Computation of H(Zk)

                if (alfa == 1.0) {
                    fauxz=fzk;
                    hauxz=hzk;
                    memcpy(constauxz,constz,m*sizeof(double)); // Value of constraints in point Zk
                }

                contalfa=contalfa+1;

                nodezk = NewNode(fzk,hzk,miu);
                pointerzk = Acceptable(nodezk,filter);
                if (pointerzk != NULL )
                    if (hzk <= (1-delta)*hxk) //Decr{\'e}scimo significativo em rela\c{c}{\~a}o ao ponto xk
                        Accept = TRUE;          //Sai do ciclo
                    else alfa = alfa/2;  //Se o decr{\'e}scimo n{\~a}o {\'e} siginificativo divide por 2
                else alfa = alfa/2;    //Se n{\~a}o {\'e} aceit{\'a}vel no filtro divide por 2
                pointerzk = NULL;

            } //End While - Feasible phase - find direction

            if ((!Accept)&&(alfa < TolAlfa))    {
                for(i=0;i<n;i++)  //use the search direction obtain in LP, with alfa=1
                    zk[i]=xk[i]+dfea[i];
                fzk=fauxz;
                hzk=hauxz;
                memcpy(constz,constauxz,m*sizeof(double)); // Value of constraints in point Zk
            }

            jacval(zk, jacobian, NULL);
            objgrd(0, zk, gradObj, NULL);
            contfeaG=contfeaG+1; //count of gradient
        } //End else of if(INFORM==2)
    } //End else of if(r==p && p==1)
} //End else of if(hxk <= zero)
//*****************END OF FEASIBILITY PHASE*****************

residuo = Convergence (gradObj, CLAMBDA, jacobian, n, m); //verificar se o ponto {\'e} estacion{\'a}rio

converge = ((residuo <= TolStop)&&(hzk<=zero));

contalfa=0;   //contador de divis{\~o}es do alfa

/*****************OPTIMALITY PHASE*****************/
if (!converge) {
    // Calculate optimality direction - QP2 Problem

    /*objgrd   -> computes the gradient, at point Xk, of objective nobj (number of OF) and stores in gradObj : nobj = 0 -> 1 O.F.
    lagscale -> definiton of factor sigma
    fullhes  -> computes Lagrangian Hessian (W) and stores it in hessianLagr*/

    fullhes(hessianLagr,n,0,NULL,lambda);
```

```
memcpy(CLAMBDAaux,CLAMBDA,(n+m)*sizeof(double));

solveLSSOLQP(m,n,lb,ub,zk,jacobian,constz,gradObj,CLAMBDA,hessianLagr,dopt,&INFORM);

//N faz optimalidade se INFORM==2
if ((INFORM==2)) {
    memcpy(xk1, zk, n*sizeof(double));
    hxk1 = hzk;
    fxk1 = fzk;
    memcpy(constx1,constz,m*sizeof(double)); // Value of constraints in point Xk1
    memcpy(CLAMBDA,CLAMBDAaux,(n+m)*sizeof(double));
}
else{
    if (TestDirection(dopt,n)!=0) { //only divides alpha if dopt!=0
        contopt=contopt+1;              //count of optimalities
        Accept = FALSE;
        alfa = 1.0;
        while ((!Accept) && (alfa >= TolAlfa)) {
            /*Obtain Xk+1 point*/
            for(i=0;i<n;i++)
                xk1[i]=zk[i]+alfa*dopt[i];

            fxk1 = objsign*objval(0,xk1,NULL);    //calculates fxk1

            contoptF=contoptF+1;                  //count of O.F. in optimality phase

            conval(xk1, constx1, NULL);           //constraints value

            hxk1 = Func_H(lb,ub,constx1,xk1,n,m); //calculates hxk1

            if (alfa == 1.0) {
                faux=fxk1;
                haux=hxk1;
                memcpy(constaux,constx1,m*sizeof(double)); // Value of constraints in point Xk1
            }

            contalfa=contalfa+1;

            nodexk1 = NewNode(fxk1,hxk1,miu);
            pointerxk1 = Acceptable(nodexk1,filter);
            if (pointerxk1 != NULL )
                if (fxk1 <= ((1-delta)*fzk))    //Decr{\'e}scimo significativo do f em rela\c{c}{\~a}o ao ponto zk
                    Accept = TRUE;
                else alfa = alfa/2;
            else alfa = alfa/2;

            pointerxk1 = NULL;

        } //Fim do While - Find direction dopt

        if ((!Accept)&&(alfa < TolAlfa))    {
            for(i=0;i<n;i++)  //use the search direction obtain in QP, with alfa=1
                xk1[i]=zk[i]+dopt[i];
            fxk1=faux;
            hxk1=haux;
            memcpy(constx1,constaux,m*sizeof(double)); // Value of constraints in point Xk1
            memcpy(CLAMBDA,CLAMBDAaux,(n+m)*sizeof(double));
        }
        else {
            for (i=0 ; i<n+m; i++)
                CLAMBDA[i] = - CLAMBDA[i];
            for (i=0 ; i<m; i++)
                lambda[i] = CLAMBDA[n+i];
        }
    }  //End if(dopt!=0)
    //***************** END OF OPTIMALITY PHASE*****************
    else {  //If(Dopt==0) -- there is no optimality phase
        memcpy(xk1, zk, n*sizeof(double));
        hxk1 = hzk;
        fxk1 = fzk;
        memcpy(constx1,constz,m*sizeof(double)); // Value of constraints in point Xk1
        //Dopt d{\'a} zero mas INFORM tb d{\'a} zero!!
        for (i=0 ; i<n+m; i++)
            CLAMBDA[i] = - CLAMBDA[i];
        for (i=0 ; i<m; i++)
            lambda[i] = CLAMBDA[n+i];
    }//End if((INFORM==2)
}//End else INFORM==2

//Filter update
//se:    fxk1 < fxk  : f-iteration - n{\~a}o faz nada, i {\'e}, n{\~a}o insere o ponto no filtro
//sen{\~a}o: fxk1 >= fxk : h-iteration - insere o par no filtro e remove os dominados
if (fxk1 >= (fxk - min(hxk*hxk,omega))) {
    pointerxk = Acceptable(nodexk,filter);
    if (pointerxk != NULL) {
        AcceptInFilter = TRUE;
        InsertUpdateFilter(nodexk,&filter,pointerxk);
    }
```

```
                else
                    AcceptInFilter = FALSE;
            }
            else
                AcceptInFilter = FALSE;

            pointerxk = NULL;

            if (AcceptInFilter)
                cont = cont+1;

            aux=filter;
            contFiltro=0;
            while (aux != NULL) {
                fprintf(fich,"auxf=%f\t auxh=%f\n",aux->f,aux->h);
                aux=aux->next;
                contFiltro=contFiltro+1;
            }
            fprintf(fich,"\n");
            fprintf(fich,"+++ Nž de elementos no filtro=%d +++\n\n",contFiltro);

            k = k+1;  //New iteration

            memcpy(xk, xk1, n*sizeof(double)); // Xk <- Xk+1
            fxk=fxk1; //f(Xk) <- f(Xk+1)
            hxk=hxk1;
            memcpy(constx,constx1,m*sizeof(double)); // Value of constraints in point Xk

            objgrd(0, xk, gradObj, NULL);   //gradient
            contG=contG+1; //count of gradient

            jacval(xk, jacobian, NULL);    //jacobian

            nodexk = NewNode(fxk,hxk,miu); //new trial pair

            residuo = Convergence(gradObj, CLAMBDA, jacobian, n, m);

            converge = ((residuo <= TolStop)&&(hxk<=zero));

        }  //End if(!converge) -> optimality phase
        else { //if converge in feasibility phase -- Found a KKT point
            k = k+1;
            memcpy(xk, zk, n*sizeof(double)); // Xk <- zk
            fxk=fzk;
            hxk=hzk;
        }
    } //End main cicle While
} //Fim if (!setjmp(Jb.jb))

end= time(&fim);          // fim da medida de tempo
telapsed= (float)(end-start)/CLK_TCK;   //converte tempo para segundos

//Prints to file
fprintf(fich,"\n\n");
fprintf(fich,"******Solu\c{c}{\~a}o Encontrada****** \n");
fprintf(fich,"+++ Ponto {\'O}ptimo +++\n");
PrintVector (xk, fich, n);
fprintf(fich,"\n");
fprintf(fich,"f= %f\n", fxk);
fprintf(fich,"h= %f\n\n", hxk);
fprintf(fich,"Residuo= %.10f\n\n", residuo);
fprintf(fich,"Nž Itera\c{c}{\~o}es= %d\n\n", k-1);
fprintf(fich,"Nž Inser\c{c}{\~o}es no filtro= %d\n", cont);
fprintf(fich,"Nž Elementos no filtro= %d\n\n", contFiltro-1);
fprintf(fich,"Nž Admissibilidades= %d\n", contfea);
fprintf(fich,"Admissibilidade: nž F's = %d\n", contfeaF);
fprintf(fich,"Admissibilidade: nž Grad's = %d\n", contfeaG);
fprintf(fich,"Nž Optimalidades= %d\n", contopt);
fprintf(fich,"Optimalidade: nž F's = %d\n\n", contoptF);
fprintf(fich,"Nž de c{\'a}lculos de Gradiente = %d\n\n", contG-1);
fprintf(fich,"Tempo de execu\c{c}{\~a}o: %f seg\n",telapsed);
fclose(fich);

// return solution to AMPL
write_sol("Solution found by FILTERPHASED", xk, NULL, &Oinfo);

ASL_free(&asl);
}
```

filterphased.h

```
#ifndef FILTERPHASED_Included
#define FILTERPHASED_Included

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<signal.h>

#include "amplinc\nlp.h"
#include "amplinc\getstub.h"

#define TRUE 1
#define FALSE 0

#define infinity 1.0e20

#define TolAlfa 1.0e-6   /* Tolerance of alfa in Line Search */
#define TolStop 1.0e-6   /* bound on stop criteria          */
#define TolIter 1000     /* bound on number of iterations    */
#define delta   1.0e-6   /* Tolerance to test feasability and
optimality    */

#define gama 1.0e-5      /* Envelope's Constants            */
#define beta (1-gama)    /* 0 < gama < beta < 1             */
#define omega 1.0e-20    /* Tolerance of H(x)               */
#define zero 1.0e-6      /* Zero = 0.000000000000000000001   */

#define u 1.0e4          /* upper bound on constraint
infeasibility */

typedef struct lnode *List; typedef struct lnode {
    double f;           /* Objective Function Value        */
    double h;           /* Constraints Function Value       */
    double miu;         /* Least power of ten larger than Infinity Norm of Lambda Coefficients */
    List next;
} Listnode;

char *filter_opt_i(Option_Info *oi, keyword *kw, char *value);
char *filter_opt_d(Option_Info *oi, keyword *kw, char *value);

struct Options {
  int exemploi;              // Exemplo inteiro
  double exemplod;           // Exemplo double
};


// Prototype ...
extern void *getmemory(int n);

extern void initDoubleVector(double *v, int n);
extern void initIntVector(int *v, int n);
extern void Sum (double *a, double *b, double *sum, int n);
extern void PrintVector (double *v, FILE *f, int n);
extern double Norm_L1 (double *v, int n);
extern double Norm_L2 (double *v, int n);
extern double Norm_Infinity (double *v, int n);

extern double Convergence (double *grad, double *lamLSSol, double * jac, int n, int m);
extern double Func_H (double *lb, double *ub, double *constr, double *xk, int n, int m);  /* Function H */
extern void FuncEvalC(int n,int m,double *lb, double *ub,double *constr, double *xk, int numlc, double *evalc, int *p, int *cp);
extern void FuncSetCVEC(int n,int m, double *evalc, double *jac,double *CVEC);
extern void FuncSetC(int n, int m, int p, double * evalc, double *jac,double *C);
extern int TestDirection(double *d, int n);
extern int numConstraints(int m, int n, double *ub, double *lb);
extern void Norm_Linf (double *v, int n, int *pos, double *norm);
extern void vectorH(double *H, double *lb, double *ub, double *constr, double *xk, int n, int m);
extern void CVECSpecial(int n, int m, int pos, double *evalc, double *jac,double *CVEC);
extern double modulo(double x);
extern void multMatriz(int l, int cc, int p, double *a, double *b, double *c);
extern void transposta(int lin, double *a, double *b);
extern void egpp(int n, double *a, double *b, double *x, int *singular);
extern void printv(double *v, int n);
extern void printint(int *v, int n);
extern void solveLSSOLP(int m,int n, int p, int r, int numlc, int cp, double *lb, double *ub, double *evalc, double *xk,double
*constx, double *jacobian, double *dfea, long* INFORM, double* CVEC, double *BL, double *BU);
extern void solveLSSOLQP(int m,int n, double *lb,double *ub, double *zk, double *jac,double *constz, double *gradObj, double
*CLAMBDA, double *hess,double *dopt, long* INFORM);
extern List NewNode (double f0, double h0, double miu0);
extern void FreeList (List *s); extern int SufRedF (List no, List s);
extern int SufRedH (List no, List s);
extern int North_West (List no, List s);
extern int South_East (List no, List s);
extern List Acceptable (List node, List s);
extern void InsertUpdateFilter (List node, List *s, List last);
#endif
```

routines.c

```
#include "filterphased.h"

/*LSSOL*/

/*
    subroutine lssol(M, N,
                     NCLIN, NROWC, NROWA,
                     C,BL, BU, CVEC,
                     ISTATE, KX, X, A, B,
                     INFORM, ITER, OBJ, CLAMDA,
                     IW, LENIW, W, LENW)

    integer M, N, NCLIN,
            NROWC, NROWA, INFORM, ITER, LENIW, LENW

    integer ISTATE(N+NCLIN), KX(N), IW(LENIW),
    real    OBJ
    real    C(NROWC,*), BL(N+NCLIN), BU(N+NCLIN),
            CVEC(*), X(N), A(NROWA,*),
            B(*), CLAMDA(N+NCLIN), W(LENW)

    Input:  M       //nž de linhas da matriz A. Nos problemas FP e LP {\'e} zero.
            N       //nž de vari{\'a}veis
            NCLIN   //nž de restri\c{c}{\~o}es lineares
            NROWC   //dimens{\~a}o de C {1,NCLIN}
            NROWA   //dimens{\~a}o de A {1,M}
            C       //matriz com os coeficientes das restri\c{c}{\~o}es. Tam: NROWC
            BL      //vector com os limites inferiores das restri\c{c}{\~o}es. Tam: N+NCLIN
            BU      //vector com os limites superiores das restri\c{c}{\~o}es. Tam: N+NCLIN
            CVEC    //vector com os coeficientes das N vari{\'a}veis da fun\c{c}{\~a}o objectivo. Tam: N

      I/O:  ISTATE  //vector de reais q indica o estado das restri\c{c}{\~o}es no inicio. N{\~a}o precisa ser inicializado. Tam: N+NCLIN
            KX      //nos problemas FP, LP, QP1, QP2, LS1, LS2 n{\~a}o precisa ser inicializado. Devolve a ordem das colunas no vector A. Tam: N
            X       //array de reais com o valor do ponto inicial / ponto {\'o}ptimo. Tam: N
            A       //matriz de reais. Problemas do tipo FP, LP n{\~a}o {\'e} utilizado. Dimensionar A[1,1]
            B       //problemas do tipo FP, LP, QP n{\~a}o {\'e} utilizado. Dimensionar B[1]

    Output: INFORM  //indica o resultado do LSSOL
            ITER    //nž de itera\c{c}{\~o}es
            OBJ     //valor da F.O.
            CLAMDA  //array de inteiros com os multiplicadores de Lagrange

    Workspace parameters
            IW      //array de inteiros com o tamanho LENIW q define o espa\c{c}o do LSSOL
            LENIW   //tamanho de IW. Tem de ser pelo menos N
            W       //array de reais com o tamanho LENW q define o espa\c{c}o do LSSOL
            LENW    //tamanho de W. Se o problema {\'e} LP e N <= NCLIN -> 2*N^2+7*N+6*NCLIN
                    //             Se o problema {\'e} LP e N>NCLIN>0  -> 2*(NCLIN+1)^2+7*N+6*NCLIN
                    //             Se o problema {\'e} LP e NCLIN = 0  -> 7*N
*/

extern void lsoptn_(); extern void lssol_(long *m, long *n,
            long *nclin, long *ldA, long *ldR,
            double *A, double *bl, double *bu, double *cvec,
            long *istate, long *kx, double *x, double * R, double *b,
            long *inform, long *iter, double *obj, double *clamda,
            long *iw, long *leniw, double *w, long *lenw );


List NewNode (double f0, double h0, double miu0) {
    List p = (List) malloc (sizeof (Listnode));
    p->f=f0;
    p->h=h0;
    p->miu=miu0;
    p->next=NULL;
    return p;
}

void FreeList (List *s) {
    List aux;

    while (*s != NULL) {
        aux = *s;
        *s = (*s)->next;
        free (aux);
    }
}

void PrintVector (double *v, FILE *f, int n) {
    int i;

    fprintf (f,"[%f",v[0]);
    for (i = 1 ; i < n; i++)
        fprintf (f,", %f", v[i]);
    fprintf (f,"]\n");
    fprintf(f,"\n");
```

```
}

void Sum (double *a, double *b, double *sum, int n) {  // sum of
two vectors of dimension n
    int i;

    for (i = 0 ; i < n; i++)
        sum[i] = a[i] + b[i];
}

double Norm_L2 (double *v, int n) {   /* L2 norm of vector v of
dimension n */
    int i;
    double norm, sum=0.0;

    for (i = 0 ; i < n; i++)
        sum = sum + (v[i] * v[i]);
    norm = sqrt (sum);
    return norm;
}

void Norm_Linf (double *v, int n, int *pos, double *norm)  //
Linfinity norm of vector v of dimension n and the index of the
constraint violated {
    int i;

    *norm=v[0];
    *pos=0;
    for (i=1;i<n;i++)
        if(v[i]>=*norm) {
            *norm = v[i];
            *pos = i;
        }

}

int SufRedF (List no, List s) {   /* Sufficient reduction on
objective function */
    int aux;

    if (s == NULL) return FALSE;
    if (no->f <= (s->f - (gama * no->h)))
        aux = TRUE;
    else
        aux = FALSE;
    return aux;
}

int SufRedH (List no, List s) {   /* Sufficient reduction on
constraints function */
    int aux;

    if (no->h <= (beta * s->h))
        aux = TRUE;
    else
        aux = FALSE;
    return aux;
}

int North_West (List no, List s) {   /* North-West Corner Rule */
    int aux;
    double miu;

    miu = 1000 * s->miu;
    if ((no->f + miu * no->h) <= (s->f + miu * s->h))
        aux = TRUE;
    else
        aux = FALSE;
    return aux;
}

int South_East (List no, List s) {   /* South-East Corner Rule */
    int aux;
    double miu;

    miu = s->miu / 1000;
    if ((no->f + miu * no->h) <= (s->f + miu * s->h))
        aux = TRUE;
    else
        aux = FALSE;
    return aux;
}

List Acceptable (List node, List s) {   /* Check if the new point
is accepted in the filter */
    List auxF, auxH, accept;
    int srF, srH;
```

```
    if (SufRedF (node, s)) {
        auxF = s->next;
        if (SufRedH (node, s))
            accept = s;
        else {
            auxH = s;
            srF = SufRedF (node, auxF);
            srH = SufRedH (node, auxF);
            while (srF && (! srH) && (auxF->next != NULL)) {
                auxH = auxF;
                auxF = auxF->next;
                srF = SufRedF (node, auxF);
                srH = SufRedH (node, auxF);
            }
            while (srF && (auxF->next != NULL)) {
                auxF = auxF->next;
                srF = SufRedF (node, auxF);
            }
            if (!srF)
                if (SufRedH (node, auxF))
                    accept = auxH;
                else
                    accept = NULL;
            else
                if (SufRedH (node, auxF))
                    accept = auxH;
                else
                    if (South_East (node, s))
                        accept = auxF;
                    else
                        accept = NULL;
        }
    }
    else
        if (SufRedH (node, s))
            accept = s;
        else
            accept = NULL;

    return accept;
}

void InsertUpdateFilter (List node, List *s, List last) {
    List auxF, auxF_prev;
    int srF;

    if ((*s) == last)
        if (SufRedF (node, last))
            if (SufRedH (node, last)) {
                *s = node;
                auxF_prev = last;
                auxF = last->next;
                srF = SufRedF (node, auxF);
                while (srF && (auxF->next != NULL)) {
                    auxF_prev = auxF;
                    auxF = auxF->next;
                    srF = SufRedF (node, auxF);
                }
                if (srF)
                    FreeList (&last);
                else {
                    node->next = auxF;
                    auxF_prev->next = NULL;
                    FreeList (&last);
                }
            }
            else {
                last = last->next;
                (*s)->next = node;
                srF = SufRedF (node, last);
                if (srF) {
                    auxF_prev = last;
                    auxF = last->next;
                    srF = SufRedF (node, auxF);
                    while (srF && (auxF->next != NULL)) {
                        auxF_prev = auxF;
                        auxF = auxF->next;
                        srF = SufRedF (node, auxF);
                    }
                    if (srF)
                        FreeList (&last);
                    else {
                        node->next = auxF;
                        auxF_prev->next = NULL;
                        FreeList (&last);
                    }
                }
                else
```

```
                    node->next = last;
            }
        else {
            node->next = *s;
            *s = node;
        }
    else
        if (last->next != NULL) {
            auxF = last->next;
            srF = SufRedF (node, auxF);
            if (srF) {
                auxF_prev = auxF;
                last->next = node;
                last = auxF;
                auxF = auxF->next;
                srF = SufRedF (node, auxF);
                while (srF && (auxF->next != NULL)) {
                    auxF_prev = auxF;
                    auxF = auxF->next;
                    srF = SufRedF (node, auxF);
                }
                if (srF)
                    FreeList (&last);
                else {
                    node->next = auxF;
                    auxF_prev->next = NULL;
                    FreeList (&last);
                }
            }
            else {
                last->next = node;
                node->next = auxF;
            }
        }
        else
            last->next = node;
}

double Convergence (double *grad, double *lamLSSol, double *jac,
int n, int m) {
    double *Num, *aux, *lamN, *lamM, *prod;
    double normNum, normGrad, soma, prodI, r;
    int i, j;

    lamN = (double *) getmemory(n*sizeof(double));
    initDoubleVector(lamN,n);
    for (i = 0 ; i < n; i++)
        lamN[i] = lamLSSol[i];   // lambda vector of variables

    lamM = (double *) getmemory(m*sizeof(double));
    initDoubleVector(lamM,m);
    for (i = 0 ; i < m; i++)
        lamM[i] = lamLSSol[n+i]; // lambda vector of constraints

    normGrad = Norm_L2 (grad, n);// Norm L2 of gradient vector

    for (i = 0 ; i < n ; i++)
        if (fabs(lamN[i]) > normGrad) normGrad=fabs(lamN[i]);


    for (j = 0 ; j < m ; j++){
        soma=0.0;
        for (i = 0 ; i < n; i++)
            soma = soma + jac[i*m+j] * jac[i*m+j];

        prodI = sqrt(soma)* fabs(lamM[j]);
        if (fabs(prodI) > normGrad) normGrad = fabs(prodI);
    }

    if (normGrad < 1.0) normGrad=1.0;

    prod = (double *) getmemory(n*sizeof(double));  // product of jacobian matrix and
    initDoubleVector(prod,n);
    for (i = 0 ; i < n; i++)
            prod[i] = 0.0;

    for (j = 0 ; j < m ; j++){/* lambda vector of constraints */
        for (i = 0 ; i < n; i++)
            prod[i] = prod[i] + jac[i*m+j] * lamM[j];
    }

    Num=(double *) getmemory(n*sizeof(double));
    aux=(double *) getmemory(n*sizeof(double));
    initDoubleVector(Num,n);
    initDoubleVector(aux,n);

    Sum(lamN,prod,aux,n);
```

```
    Sum(grad,aux,Num,n);

    normNum=Norm_L2 (Num, n);

    r = normNum / normGrad;

    free(aux);
    free(Num);
    free(lamN);
    free(prod);
    free(lamM);

    return r;
}

void *getmemory(int n) {
    void *p=malloc(n);
    if (p==NULL) puts("Memory alocation error"), exit(1);
    return p;
}

void initDoubleVector(double *v, int n) {
    int i;
    for (i=0;i<n;i++)
        v[i]=0.0;
}

void initIntVector(int *v, int n) {
    int i;
    for (i=0;i<n;i++)
        v[i]=0;
}

//Determinar o n{\'u}mero de restri\c{c}{\~o}es verdadeiras, i{\'e}, o lb e ub <>s de infinito
int numConstraints(int m, int n, double *ub, double *lb) {
    int i,r=0;

    for(i=0;i<n+m;i++)
        if(lb[i]>-infinity)
            r++;
        else if (ub[i]<infinity)
            r++;
    return r;
}

void solveLSSOLP(int m, int n, int p, int r, int numlc, int cp,
double *lb,double *ub, double *evalc, double *xk,double
*constx,double *jacobian,double *dfea, long* INFORM, double *CVEC,
double *BL, double *BU) {
    int i, pos=0;
    double *CLAMBDA, *W, *A, *B, *H;
    long NCLIN, NROWC, NROWA, LENIW, LENW;
    long *IW, *ISTATE, *KX;
    long ITER;
    double OBJ, norm=0.0;
    long M;
    int aux=0;

    *INFORM=0;

    if (p==0) {          //there aren't constraints violated, so the problem for LPSOL is FP type (without objective function)
        //Inicializa\c{c}{\~a}ode vectores auxiliares
        CVEC = (double *) getmemory(1*sizeof(double)); //vector que cont{\'e}m os coeficientes da F.O.
    }
    else  {//CASE 3: there are simple bounds and C(x) constraints violated
        initDoubleVector(CVEC,n);   //Inicializa\c{c}{\~a}ode vector CVEC
        if (r>p)   //Normal case: number of constraints is different of number of violated constraints
            FuncSetCVEC(n,m,evalc,jacobian,CVEC); //fun\c{c}{\~a}o que calcula CVEC - restri\c{c}{\~o}es C(x) que s{\~a}o violadas e simple bound violadas
        else if(r==p && p>1) {    //Number of constraints is equal to number of violated constraints and those are greater than 1
                                  //select the one with greater violation
                H = (double *) getmemory((n+m)*sizeof(double)); //vector with the value of each constraint violation
                initDoubleVector(H,n+m);  //Inicializa\c{c}{\~a}ode vector H
                vectorH(H,lb,ub,constx,xk,n,m);
                Norm_Linf(H,n+m,&pos,&norm);  //select the constraint with greater violation
                CVECSpecial(n,m,pos,evalc,jacobian,CVEC); //fun\c{c}{\~a}o que calcula CVEC - apenas com a restri\c{c}{\~a}o que viola mais
                aux=1;
        }
    }

    //Setting variables to LSSOL
    NROWA = 1;
    NCLIN = numlc; //nž de restri\c{c}{\~o}es lineares
    NROWC = m; //dimens{\~a}o do jaconiano;
    CLAMBDA = (double *) getmemory((n+m)*sizeof(double));
    ISTATE = (long *) getmemory((n+m)*sizeof(long));
    LENW  = 2*n*n+7*n+6*NCLIN;
    W  = (double *) getmemory(LENW*sizeof(double));
    LENIW = n;
```

```
IW = (long *) getmemory(LENIW*sizeof(long));
A = (double *) getmemory(1*sizeof(double));
B = (double *) getmemory(1*sizeof(double));
KX = (long *) getmemory(n*sizeof(long));//NULL;
M = 0;

//Initialize Vectors
initDoubleVector(BL,n+m);
initDoubleVector(BU,n+m);
initDoubleVector(CLAMBDA,n+m);
initIntVector(ISTATE,n+m);
initDoubleVector(W,LENW);
initIntVector(IW,LENIW);
initDoubleVector(A,1);
initDoubleVector(B,1);
initIntVector(KX,n);

//Os vectores BL e BU s{\'o} t{\^e}m os limites das restri\c{c}{\~o}es que n{\~a}o s{\~a}o violadas e das simple bounds
//Os primeiros n registos do lb e ub s{\~a}o copiados directamente (simple bounds) + os registos das restri\c{c}{\~o}es C(x) que n{\~a}o s{\~a}o violadas

//New bounds for LP
memcpy(BL, lb, (n+m)*sizeof(double));
memcpy(BU, ub, (n+m)*sizeof(double));

for(i=0;i<n;i++){          // simple bounds are copied directaly
    if (evalc[i] == 0) {
        if(lb[i]>-infinity)
            BL[i]=lb[i]-xk[i];
        if(lb[i]==-infinity) BL[i]=lb[i];
        if(ub[i]<infinity)
            BU[i]=ub[i]-xk[i];
        if(ub[i]==infinity) BU[i]=ub[i];
    }
    else if (aux==0){
        if (evalc[i] == -1){   //se SB {\'e} violada coloco-a livre (infinity) porque j{\'a} vai para a F.O.
            if(lb[i]>-infinity)
                BL[i] = -infinity;
        }
        else if (evalc[i] == 1){
            if(ub[i]<infinity)
                BU[i] = infinity;
        }
    }
}

for(i=n;i<n+m;i++) {         // C(x) only not violated constraints
    if (evalc[i] == 0) {
        if(lb[i]>-infinity)
            BL[i]=lb[i]-constx[i-n];
        else
            BL[i]=lb[i];
        if(ub[i]<infinity)
            BU[i]=ub[i]-constx[i-n];
        else
            BU[i]=ub[i];
    }
    else if (aux==0){
        if (evalc[i] == -1) { //violada {\`a} esquerda - coloco-a livre (infinity) porque j{\'a} vai para a F.O.
               if(lb[i]>-infinity)
                  BL[i]=-infinity;
        }
        else if (evalc[i] == 1) {  //violada {\`a} direita - o ub fica infinity
            if(ub[i]<infinity)
                BU[i]=infinity;
        }
    }
}

if (r==p && p>1) {   //Number of constraints is equal to number of violated constraints and those are greater than 1
                     //select the one with greater violation
    if (lb[pos]>-infinity)
        BL[pos] = -infinity;
    else if (ub[pos]<infinity)
        BU[pos] = infinity;
    free(H);
}

//Initializing the search direction
for(i=0;i<n;i++)
    dfea[i]=1.0;

//Get direction from LSSOL
lsoptn_("Problem Type LP", 15); //Problema do tipo LP
lsoptn_("Feasibility tolerance = 1.0e-12", 31);
lssol_(&M, &n, &NCLIN, &NROWC, &NROWA, jacobian, BL, BU, CVEC,
       ISTATE, KX,  dfea, A, B, INFORM, &ITER, &OBJ, CLAMBDA, IW,
       &LENIW, W, &LENW);
```

```
    //Free dos vectores que j{\'a} n{\~a}o s{\~a}o precisos
    free(CLAMBDA);
    free(ISTATE);
    free(W);
    free(IW);
    if (A!=NULL) free(A);
    if (B!=NULL) free(B);
    if (KX!=NULL) free(KX);

}

void solveLSSOLQP(int m,int n, double *lb,double *ub,double *zk,
double *jac,double *constz, double *gradObj, double *CLAMBDA,
double *hess,double *dopt, long* INFORM) {
    int i;

    double *QPlb, *QPub, *W, *B;
    long NCLIN, NROWC, NROWA, LENIW, LENW;
    long *IW, *ISTATE, *KX;
    long ITER;
    double OBJ;

    *INFORM=0;

    //Inicializa\c{c}{\~a}o das vari{\'a}veis para o LSSOL
    NCLIN = m;
    NROWC = m;
    NROWA = n;
    LENIW = n;
    LENW  = 2*n*n+10*n+6*m;
    IW = (long *) getmemory(LENIW*sizeof(long));
    W  = (double *) getmemory(LENW*sizeof(double));
    ISTATE = (long *) getmemory((n+m)*sizeof(long));
    KX = (long *) getmemory(n*sizeof(long));
    B = (double *) getmemory(1*sizeof(double));

    QPlb = (double *) getmemory((n+m)*sizeof(double));
    QPub = (double *) getmemory((n+m)*sizeof(double));

    /*Initialize Vectors*/
    initDoubleVector(QPlb,n+m);
    initDoubleVector(QPub,n+m);
    initIntVector(ISTATE,n+m);
    initDoubleVector(W,LENW);
    initIntVector(IW,LENIW);
    initDoubleVector(B,1);
    initIntVector(KX,n);

    memcpy(QPlb, lb, (n+m)*sizeof(double));
    memcpy(QPub, ub, (n+m)*sizeof(double));

    for(i=0;i<n;i++){ // new bounds for QP
        if(lb[i]>-infinity)
            QPlb[i]=lb[i]-zk[i];
        if(ub[i]<infinity)
            QPub[i]=ub[i]-zk[i];
    }
    for(i=0;i<m;i++){ //
        if(lb[n+i]>-infinity)
            QPlb[n+i]=lb[n+i]-constz[i];
        if(ub[n+i]<infinity)
            QPub[n+i]=ub[n+i]-constz[i];
    }

    /*Initializing the search direction*/
    for(i=0;i<n;i++)
        dopt[i]=1.0;

    lsoptn_("Problem Type QP2", 16); /* Problema do tipo QP2 */
    lssol_(&n, &n, &NCLIN, &NROWC, &NROWA, jac, QPlb, QPub, gradObj,
           ISTATE, KX,  dopt, hess, B, INFORM, &ITER, &OBJ, CLAMBDA, IW,
           &LENIW, W, &LENW);


    //Free dos vectores que j{\'a} n{\~a}o s{\~a}o precisos
    free(ISTATE);
    free(QPlb);
    free(QPub);
    free(W);
    free(IW);
    free(B);
    free(KX);
}

double Func_H (double *lb, double *ub, double *constr, double *xk,
int n, int m) /* Function H */ {
    double h=0.0, auxh=0.0;
    int i;
```

```
        for(i=0;i<n;i++){
            if(lb[i]>-infinity && (auxh=lb[i]-xk[i])>zero)
                h+=auxh;
            if(ub[i]<infinity && (auxh=xk[i]-ub[i])>zero)
                h+=auxh;
        }
        for(i=0;i<m;i++){
            if(lb[n+i]>-infinity && (auxh=lb[n+i]-constr[i])>zero)
                h+=auxh;
            if(ub[n+i]<infinity && (auxh=constr[i]-ub[n+i])>zero)
                h+=auxh;
        }

    return h;
}

//Function to determine the violation of each constraint - Vector H
void vectorH(double *H, double *lb, double *ub, double *constr,
double *xk, int n, int m) {
    int i;

    for (i=0; i<n; i++) {
        if(lb[i]>-infinity && (lb[i]-xk[i])>zero)  //value of sb left violation
            H[i] = lb[i]-xk[i];
        else if(ub[i]<infinity && (xk[i]-ub[i])>zero)  //value of sb right violation
            H[i] = xk[i]-ub[i];
        else
            H[i] = 0.0;
    }
    for (i=n; i<n+m; i++) {
        if(lb[i]>-infinity && (lb[i]-constr[i-n])>zero)  //value of C(x) left violation
            H[i] = lb[i]-constr[i-n];
        else if(ub[i]<infinity && (constr[i-n]-ub[i])>zero)  //value of C(x) right violation
            H[i] = constr[i-n]-ub[i];
        else
            H[i] = 0.0;
    }
}

/* Function that evaluates wich constraints are violated for a
given point Xk
   The number of violated constraints is set on p;
   Vector evalc stores wich constraints are violated;
   Index 0..n evaluates the simple bounds constraints and the index n..m (n+i) evaluates the C(x) constraints.

   Not violated      -> evalc = 0
   Violated Direita  -> evalc = 1
   Violated Esquerda -> evalc = -1
*/

void FuncEvalC(int n,int m,double *lb, double *ub,double *constr,
double *xk, int numlc, double *evalc, int *p, int *cp) {
    int i, j=0;

    *p=0;
    *cp=0;
    for (i=0; i<n; i++) {
        if(lb[i]>-infinity && (lb[i]-xk[i])>zero) { //simple bound is violated {\`a} esquerda
            evalc[i] = -1.0;
            (*p)++;
        }
        else
            if(ub[i]<infinity && (xk[i]-ub[i])>zero) { //simple bound is violated {\`a} direita
                evalc[i] = 1.0;
                (*p)++;
            }
            else    //simple bound is not violated
                evalc[i] = 0.0;
    }

    for (i=n; i<n+m; i++) {
        if(lb[i]>-infinity && (lb[i]-constr[i-n])>zero) { //constraint is violated
                evalc[i] = -1.0; //violada {\`a} esquerda
                (*p)++;
                (*cp)++; //apenas restri\c{c}{\~o}es c(x) violadas
            }
            else
                if(ub[i]<infinity && (constr[i-n]-ub[i])>zero) { //constraint is violated
                    evalc[i] = 1.0; //violada {\`a} direita
                    (*p)++;
                    (*cp)++; //apenas restri\c{c}{\~o}es c(x) violadas
                }
                else        //constraint is not violated
                    evalc[i] = 0.0;
    }
}
```

```
void FuncSetCVEC(int n, int m, double *evalc, double *jac,double
*CVEC) {
    int i, j;
    double aux;

    for (j=0; j<n; j++){
        aux = 0.0;
        for (i=n; i<n+m; i++)
            aux = aux + jac[i-n+j*m]*evalc[i];

        CVEC[j] = aux + evalc[j];
    }
}

void CVECSpecial(int n, int m, int pos, double *evalc, double
*jac,double *CVEC) {
    int j;

    if (pos<n) //the constraint violated is a simple bound
        for (j=0; j<n; j++)
            if (j==pos)
                CVEC[j] = evalc[pos];
            else
                CVEC[j] = 0.0;
    else    //the constraint violated is C(x)
        for (j=0; j<n; j++)
            CVEC[j] = jac[m*j+pos-n]*evalc[pos];
}

int TestDirection(double *d, int n) {
    int i;
    int res = 0;

    for (i=0;i<n;i++)
        if (d[i] != 0.0)
            res = res + 1;
    return res;
}
```