PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Universidade Federal do Rio de Janeiro

DISSERTAÇÃO DE MESTRADO

UMA ESTRATÉGIA HÍBRIDA PARA O MÉTODO DOS GRADIENTES CONJUGADOS NÃO LINEAR

Leonardo Castro da Silva

Instituto de Matemática

Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais

Leonardo Castro da Silva

UMA ESTRATÉGIA HÍBRIDA PARA O MÉTODO DOS GRADIENTES CONJUGADOS NÃO LINEAR

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Matemática, Instituto Tércio Pacitti (iNCE), Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Ciência em Informática.

Orientadora: *Prof^a*. D.Sc. Luziane Ferreira de Mendonça (UFRJ)

Rio de Janeiro
Maio de 2011

FICHA CATALOGRÁFICA PREPARADA PELA BIBLIOTECA DO NÚCLEO DE COMPUTAÇÃO ELETRÔNICA

Silva, Leonardo Castro da.
S586 Uma estratégia híbrida para o método dos gradientes conjugados não-linear / Leonardo Castro da Silva - Rio de Janeiro, UFRJ, 2011.
181 f.: il.

Dissertação (Mestrado em Informática) - Universidade Federal do Rio de Janeiro. Instituto de Matemática, Núcleo de Computação Eletrônica, 2011.

Orientadora: Luziane Ferreira de Mendonça.

1 - Otimização Irrestrita. 2 - Programação Não-Linear.
3 - Método dos Gradientes Conjugados - Teses. I. Luziane Ferreira de Mendonça (Orient.). II. Universidade Federal do Rio de Janeiro, Instituto de Matemática, Núcleo de Computação Eletrônica.
III. Título.

CDD

Uma Estratégia Híbrida para o Método dos Gradientes Conjugados Não Linear

Leonardo Castro da Silva

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti (iNCE) da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado por:

Profa. Dra. Luziane Ferreira de Mendonça (Orientadora)

Profa. Dra. Juliana Vianna Valério

Profa. Dra. Marcia Helena da Costa Fampa

Rio de Janeiro, Maio de 2011.

Dedicatória

Dedico este trabalho a um dos pilares de minha vida: família. Muito obrigado, mãe, pai, irmã e sobrinho.

Agradecimentos

Agradeço primeiramente ao Deus maior. A Ele todo o meu agradecimento.

Sem dúvida, à minha querida orientadora: Luziane. A quem devo por muita paciência, dedicação, presença, conselho e paciência (de novo). Sempre perfeita na hora de ajudar e também "puxar as orelhas".

Aos amigos de turma e luta: Ueverton, Juliana, Adriana, Murilo e Carlos Felipe.

Aos amigos ausentes, mas sempre presentes: Tiago Daniel, Alessandra, Luisy, Cláudio, Pablo, Douglas e Miguel.

Os meus agradecimentos também, aos professores e funcionários do PPGI/UFRJ.

Meus agradecimentos a CAPES, responsável pelo incentivo financeiro que com certeza foi de grande importância.

Resumo

SILVA, Leonardo Castro da. **Uma Estratégia Híbrida para o Método dos Gradientes Conjugados Não Linear**, 2011. 179 p. Dissertação (Mestrado em Informática), Instituto de Matemática, Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, 2011.

O método dos Gradientes Conjugados teve sua criação em 1952, quando Hestenes & Stiefel [1] o propuseram para resolver problemas de minimização especificamente sob a forma de funções quadrática. Mais tarde Fletcher & Reeves [2] estenderam seu algoritmo para buscar minimizadores nos problemas também não quadráticos, e então a otimização não-linear irrestrita ganhava para a sua área mais um método que propunha resolver problemas não-lineares.

No método dos Gradientes Conjugados as variações que são feitas no algoritmo e têm maior significância em desempenho computacional são as alterações na forma de cálculo do escalar β, que está diretamente ligado a busca das novas direções de descida a cada iteração.

Nesta dissertação propomos um estudo comparativo de desempenho do método dos Gradientes Conjugados implementado com os $\beta's$ mais utilizados na literatura, quando aplicado a um conjunto tradicional de problemas-teste de minimização irrestrita [3].

Com base nesse estudo, realizamos uma análise teórica de um novo β aqui proposto, o qual também será submetido aos mesmos testes numéricos.

Abstract

SILVA, Leonardo Castro da. **Uma Estratégia Híbrida para o Método dos Gradientes Conjugados Não Linear**, 2011. 179 p. Dissertação (Mestrado em Informática), Instituto de Matemática, Núcleo de Computação Eletrônica, Universidade Federal do Rio de Janeiro, 2011.

The conjugate gradient method had its inception in 1952, when Hestenes & Stiefel [1] deminimização proposed specifically to solve problems in the form of quadratic functions. Fletcher & Reeves [2] later extended their algorithm to seek to minimize the problems do not quadratic, then the unconstrained nonlinear optimization for your area gained over a proposed method to solve nonlinear problems.

In the method of conjugate gradient changes that are made in the algorithm and have greater significance in computational performance are changes in the calculation of the scalar $beta$, which is directly connected to the search for new directions of descent at each iteration.

In this paper we propose a comparative study of performance of the conjugate gradient method implemented with the $beta'$ s most widely used in literature, when applied to a traditional set of test problems for unconstrained minimization [3].

Based on this study, we conducted a theoretical analysis of a new $beta$ proposed here, which will also be subject to the same numerical tests.

Notações e Definições

É preciso antes de tudo definir matematicamente alguns termos que iremos utilizar ao longo deste trabalho.

Podemos definir um problema de otimização como sendo:

$$\begin{aligned} & min\ f(x) \\ s.a\quad & h_i(x) = 0, \quad i \in D_1 \\ & g_i(x) \leq 0, \quad i \in D_2 \\ & l_i(x) \geq 0, \quad i \in D_3 \\ & x \in \Re. \end{aligned} \tag{1}$$

em que, $f(x)$ é a função objetivo; h_i, g_i e l_i são funções de restrições; D_1, D_2 e D_3 são os conjuntos de índices das restrições; x é um vetor de variáveis.

Definição 1. ***Otimização irrestrita*** *é uma classe da otimização cujos problemas são aqueles em que $D_1 = D_2 = D_3 = \emptyset$, ou seja, a função objetivo não depende de restrições, pois elas não comprometem a solução ou o problema foi reescrito fazendo com que as restrições sejam substituídas por funções de penalidades.*

Definição 2. ***Programação Não-Linear*** *é o estudo de problemas em que ou pelo menos uma das restrições ou a função objetivo são não-lineares.*

Definição 3. ***Otimização determinística*** *examina o domínio não como um todo e sim apenas em uma região, ou seja, o modelo (problema) é completamente conhecido, pois se tem o conhecimento das derivadas da função objetivo.*

Definição 4. *Uma* ***função*** *é dita* ***suave*** *caso ela seja contínua e diferenciável. Assumimos que a função f a ser minimizada é duas vezes diferenciável, apesar de que é possível exigir somente que f tenha apenas gradiente contínuo para conseguir bons resultados de convergência [4].*

Definição 5. *Um conjunto $D \subset \Re^n$ é chamado* ***conjunto convexo*** *se para quaisquer $x \in D$, $y \in D$ e $\alpha \in [0,1]$, tem-se que $\alpha x + (1-\alpha)y \in D$.*

Definição 6. *Se $D \subset \Re^n$ é um conjunto convexo, uma função $f : D \to \Re$ é dita* ***convexa*** *em D quando para quaisquer $x \in D$, $y \in D$ e $\alpha \in [0,1]$, tem-se que*

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y). \tag{2}$$

Definição 7. *A função f diz-se* ***estritamente convexa*** *quando a desigualdade (2) é estrita para todo $x \neq y$ e todo $\alpha \in (0,1)$.*

Propriedade 1. *Teorema: Sejam $D \subset \Re^n$ um conjunto convexo e $f : D \to \Re$ uma função convexa em D.*
Então todo minimizador local em problema 1 ($D_1 = D_2 = D_3 = \emptyset$) é global. Além disso, o conjunto de minimizadores é convexo.
Se f estritamente convexa, não pode haver mais de um minimizador.

Definição 8. *Se $f : \Re^n \mapsto \Re$ é diferenciável, então o gradiente ∇f da função é definido [5] por*

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x) \end{bmatrix}.$$

Definição 9. *Se f é duas vezes diferenciável, ou seja, se ∇f é diferenciável então definimos*

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \frac{\partial^2 f}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

A matriz $\nabla^2 f(x)$ é chamada de matriz Hessiana [5].

Definição 10. *Seja $||.||$ uma norma no $\Re^n$. A norma induzida da matriz A do $N \times N$ é definida como sendo*

$$||A|| = \max_{||x||=1} ||Ax||.$$

Propriedade 2. *Uma importante propriedade da norma induzida*

$$||Ax|| \leq ||A||||x||.$$

Definição 11. *O número condicional da matriz A em relação a norma $||.||$ é expresso da seguinte forma*

$$\kappa(A) = ||A||||A^{-1}||.$$

Deve-se observar que a dificuldade para encontrar a solução de um determinado problema $f(x)$, é diretamente relacionado com o κ sobre a sua matriz *Hessiana* - $\kappa(\nabla^2 f(x))$.

Definição 12. *Definimos como a norma Euclidiana*

$$||x||_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}.$$

Definição 13. *Uma matriz A é dita* ***definida positiva*** *se* $x^T Ax > 0,\ \forall x \in \Re^n\, x \neq 0$.

Definição 14. *Mínimo Local: Se* $x^* \in S$ *é um mínimo local de f sobre S, então existe um* $\delta > 0$, *onde* $f(x) \geq f(x^*)\ \forall x \in S$, *tal que* $|x - x^*| < \delta$.

Definição 15. *Mínimo Global: Se* $x^* \in S$ *é um mínimo global de f sobre S, então existe um* $\delta > 0$, *onde* $f(x) \geq f(x^*)\ \forall x \in S$, *tal que* $x \neq x^*$.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O Método dos Gradientes Conjugados consiste num método iterativo designado a resolução de problemas matemáticos não-lineares. Tal método está incluído na classe de problemas de otimização irrestrita e tem como características o baixo custo computacional (visto que o cálculo de sua Hessiana não é necessário) e convergência global, sob certas hipóteses.

É um método que baseia-se na estratégia de buscar o minimizador da função nas direções ortogonais conjugadas acrescentando uma dimensão a cada iteração.

No Capítulo 2 descrevemos o método dos Gradientes Conjugados em seu formato padrão, o qual se destina à resolução de problemas de minimização irrestritos quadráticos, ou seja, tinha a principal finalidade de resolver *sistemas lineares*.

Também apresentamos um estudo da *análise de convergência* do método juntamente com suas propriedades, concentrado em problemas sob a forma de quadrática.

O Capítulo 3 traz o Método dos Gradientes Conjugados designado para resolver problemas matemáticos não-quadráticos (o método generalizado, centrado na resolução de problemas não-quadráticos). A estrutura do seu algoritmo é igual ao do método quando aplicado a problemas quadráticos, sendo que existem diferentes possibilidades de escolha

para o parâmetro β (para o caso de problemas quadráticos isso não ocorre um vez que a escolha do β é única).

Neste capítulo trazemos também a *análise de convergência* para o método aplicado aos problemas não-quadráticos.

O Capítulo 4 mostra os resultados e discussões dos diversos testes que fizemos com duas expressões para β muito estudadas na literatura: o proposto por Fletcher e Reeves [2] e o proposto por Polak e Ribière [6].

No Capítulo 5 é onde apresentamos a estratégia do novo β proposto neste trabalho de dissertação, explorando a ideia e apresentando também duas possíveis escolhas de um parâmetro que está ligado diretamente ao controle das novas direções d_k.

Finalmente, no último capítulo apresentamos os resultados dos testes feitos com os mesmos problemas do Capítulo 4, mas com o novo β proposto, incluindo também a análise comparativa entre os β's utilizados, utilizando gráficos de perfis de desempenho.

Capítulo 2

Método dos Gradientes Conjugados

2.1 Histórico

Os primeiros escritos sobre o método dos Gradientes Conjugados são datados de 1951, quando Stiefel o expos num simpósio na Universidade da Califórnia em Los Angeles, enquanto paralelamente Hestenes trabalhava no Instituto de Análise Numérica, do National Bureau of Standards. Mas foi só em 1952 que Magnus R. Hestenes e Eduard Stiefel resolveram se unir e iniciaram o trabalho em conjunto, com a descrição do que viria a ser o primeiro trabalho versando sobre o método[1] [1]. Surgia então o método dos Gradientes Conjugados que iterativamente buscava a solução de sistemas lineares, $Ax = b$ com n equações lineares e n incógnitas, terminando em no máximo n passos[2]. Os algoritmos desta classe eram destinados, essencialmente, a encontrar um elipsóide n dimensional, isto é, acometido a problemas quadráticos. Hestenes e Stiefel diziam que para encontrar soluções para estes problemas o método de eliminação de Gauss e o método dos Gradientes Conjugados eram os mais indicados por melhor se encaixar em algumas propriedades e características, tais como: simplicidade, baixo custo computacional, rápida convergência,

[1] Publicado em um artigo para o Journal of Research - National Bureau of Standards.
[2] Caso erros de arredondamentos não fossem considerados.

diminuição de erros de arredondamento e que cada passo da iteração deve estimar uma nova solução melhor que a anterior.

Contudo, mais tarde, o método foi estendido para casos também não quadráticos. Assim, em 1964, R. Fletcher e C. M. Reeves [2] modificaram pela primeira vez o método dos Gradientes Conjugados, introduzindo a idéia de incluir uma combinação linear de direções, ou seja, buscar outras novas direções e também terminar esta busca somente quando algum critério de parada fosse alcançado. O conceito de terminar a busca da solução somente quando algum critério é atingido se dá pelo fato de que o método dos Gradientes Conjugados quando aplicado à problemas não quadráticos pode não terminar no $n-$ésimo passo[3].

Em métodos de aproximações quadráticas realizam-se aproximações em x_k fazendo a substituição do resíduo do sistema linear pelo gradiente da função $f(x)$ [7]. Nos casos de problemas não quadráticos deve-se ressaltar que as funções deste tipo podem ser aproximadas, sucessivamente, por quadráticas; porém calcular a hessiana a cada iteração é computacionalmente muito caro. Deste modo, a principal mudança que podemos mencionar de Fletcher e Reeves para Hestenes e Stiefel é que a partir deles não se faz mais necessário a estimativa de $\nabla^2 f(x)$ a cada passo (característica esta, às vezes, impraticável) e assim fazer a desejável eliminação de $\nabla^2 f(x)$ sendo que, como se trata de problemas não quadráticos, desta vez é requerido uma busca linear[4]. É indispensável situar que a exatidão da busca linear é um fator crítico do método. Seja como for, o método dos Gradientes Conjugados depende apenas dos valores da função $f(x)$ e de seu gradiente [5].

Em 1969, E. Polak e G. Ribière [6] deram suas contribuições ao método, que se define mais eficiente numericamente comparado ao Fletcher-Reeves [8]. A versão de Polak e Ribière para o método dos Gradientes Conjugados não se distancia muito de Fletcher-

[3]Mais a frente, veremos que para o caso de funções quadráticas com n variáveis, a busca do minimizador desta função é feita até no máximo a $n-$ésima iteração.

[4]Visto que para quadrática temos a busca linear exata.

Reeves, difere somente no modo de encontrar o escalar β_k. Na fórmula que Fletcher e Reeves propuseram, muitas das vezes, a conjugação das direções progressivamente poderia ir se perdendo, não só pela imprecisão da busca linear, mas também pelos termos da função não quadrática, ou seja, com Fletcher-Reeves a relevante característica da descendência $d_k^T \nabla f(x_k) \leq 0$ nem sempre pode ser mantida na obtenção da nova direção de busca de d_k. Por isso a fórmula para β_k de Polak & Ribière é uma possibilidade para contornar este problema que, em geral, tem um melhor desempenho quando aplicado em funções não quadráticas [9].

Muitos pesquisadores nas últimas décadas têm se dedicado ao desenvolvimento de outros métodos numéricos e muitos acreditam que o método dos Gradientes Conjugados se encontra em um estágio de estagnação. Entretanto, é possível observar que atualmente o método dos Gradientes Conjugados ainda é frequentemente utilizado na resolução dos mais diversos problemas tais como: dinâmica não-Linear de estruturas [10], redes neurais [11], geofísica [12], a solução inversa no método de regularização [13], etc.

Esse fato é melhor ilustrado utilizando-se uma busca em site especializado em trabalhos científicos; de acordo com a ferramenta *ISI Web of Knowlegde*[5] (*Web of Science*) com as palavras-chave "Conjugate" e "Gradient", foram listados mais de 3.000 citações de trabalhos publicados nos últimos 10 anos. Muitos pesquisadores de diversos partes do mundo fizeram ou ainda fazem estudos com o método, e a Tabela 2.1 apresenta as citações para cada triênio.

Web of Science - Citações	
Anos	N° de Citações
2010-2007	237
2007-2004	164
2004-2001	141

Tabela 2.1: Número de citações a cada triênio.

[5] www.isiknowledge.com

2.2 O Método

O método Gradientes Conjugados é um método que pertence a uma classe de algoritmos de otimização irrestrita que são caracterizados pelo baixo custo computacional com forte convergência local e global [14]. Tal método baseia-se na busca da solução através de sucessivas direções conjugadas. Direções estas que são determinadas ao longo de cada passo das iterações [7], ou seja, sua essência é buscar a solução nas direções ortogonais conjugadas $-\nabla f(x_k)$, acrescentando assim uma dimensão, a cada iteração, na procura pelo minimizador. Esta família tem como componente elementar o gradiente ∇f da função objetivo f, ou seja, fazemos uso de condições como as de primeira e também de segunda ordem para a função f, em que aproximamos a função f por uma função quadrática, determinada pela série de Taylor.

2.3 Problemas Quadráticos

O método dos Gradientes Conjugados como mencionado anteriomente foi proposto originalmente para resolução de problemas quadráticos.

$$minimizar \quad \{f(x) = \frac{1}{2}x^T Ax - b^T x + c \mid x \in \Re^n\}, \tag{2.1}$$

em que A é uma matriz $n \times n$ definida positiva, b é um vetor do $\Re^n$ e c um escalar do $\Re$.

Os vetores gradientes da função objetivo $g_0, \ldots, g_k$ em cada iteração x_k são denotados por

$$g_k = Ax_k - b,$$

e são ortogonais entre si.

O problema sob a forma de quadrática nos dá uma única solução, e é exatamente o ponto crítico buscado na solução do sistema linear $Ax = b$ ($g_k = 0$). Deste modo, define-se o método dos Gradientes Conjugados por buscar "ponto-a-ponto" o ponto crítico (como A é definida positiva, a solução encontrada será o minimizador de f) de $f(x)$ ao longo da direção d_k obtida no espaço gerado por $g_0, \ldots, g_k$. E uma vez encontrada a direção, é possível calcular o tamanho do passo α_k e realizar o deslocamento

$$x_{k+1} = x_k + \alpha_k d_k.$$

Este procedimento é repetido, iteração à iteração, até que x_k esteja suficientemente próximo da solução.

Cabe ressaltar que diversos problemas de otimização não necessariamente trazem uma matriz A definida positiva. Mesmo assim, o método dos Gradientes Conjugados pode ser usado para resolver sistemas que apresentem tais matrizes [15].

2.3.1 Algoritmo Gradientes Conjugados

Para qualquer ponto $x_0 \in \Re^n$ que iniciarmos o algoritmo do método dos Gradientes Conjugados, a solução do problema converge para o único x^* e este é o minimizador global, desde que a matriz A seja definida positiva (ver Teorema Convergência 2.3).

O tamanho do passo α_k é determinado pela busca linear exata. Ele será obtido de modo gerar a menor imagem ao longo da direção d_k, ou seja, é aquele que,

$$\text{minimiza } f(x_k + \alpha_k d_k).$$

Quando aplicamos diretamente na função $f(x)$ temos,

$$\min f(x_k + \alpha_k d_k) = \min \frac{1}{2}(x_k + \alpha d_k)^T A(x_k + \alpha d_k) + b^T(x_k + \alpha d_k) + c$$

O tamanho do passo α é encontrado quando escrevemos a condição de otimalidade de primeira ordem para o problema anterior, ou seja,

$$\begin{aligned}
\frac{1}{2}[\, x_k^T A d_k + d_k^T A x_k + 2\alpha d_k^T A d_k \,] + b^T d_k &= 0 \\
d_k^T A x_k + \alpha d_k^T A d_k + b^T d_k &= 0 \\
\alpha d_k^T A d_k &= -d_k^T A x_k - b^T d_k \\
\alpha &= -\frac{d_k^T A x_k - b^T d_k}{d_k^T A d_k} \\
\alpha &= -\frac{d_k^T (A x_k - b^T)}{d_k^T A d_k} \\
\alpha &= -\frac{d_k^T g_k}{d_k^T A d_k}. \qquad (2.2)
\end{aligned}$$

A direção d_k por onde é deslocado o ponto x_{k+1} será definida mais adiante. Antes precisamos apresentar o lema e a definição seguintes, pois fazemos uso deles para obter a fórmula de d_k.

O lema a seguir mostra que o vetor gradiente da iteração $k+1$ é ortogonal às direções d das iterações anteriores.

Lema 2.1. *No método dos Gradientes Conjugados, os gradientes g_k, com $k = 0, \ldots, n$ satisfazem*

$$g_{k+1}^T d_j = 0 \quad \forall \quad j \leq k \; .$$

Demonstração. Seja $d_k = x_{k+1} - x_k$, onde x_{k+1} foi o minimizador de $f(x)$ no espaço gerado por $g_0, \ldots, g_k$. Sabemos também que x_k foi o minimizador de $f(x)$ no espaço gerado por

$g_0, \ldots, g_{k-1}$. Assim, como $d_k \in span\{g_0, \ldots, g_k\}$, podemos escrevê-lo como

$$d_k = \sum_{i=0}^{k} \gamma_i g_i.$$

Como g_{k+1} e g_j são ortogonais para $j = 0, \ldots, k$, então

$$\begin{aligned} g_{k+1}^T d_j &= g_{k+1}^T \left(\sum_{i=0}^{k} \gamma_i g_i \right) \\ &= \sum_{i=0}^{k} \gamma_i g_{k+1}^T g_i \\ &= 0 \qquad \forall \ j = 0, \ldots, k. \end{aligned}$$

■

Definição 2.1. *O conjunto de vetores não-nulos $d_0, \ldots, d_k \in \Re^n$ são A-conjugados [9], se*

$$d_i^T A d_j = 0 \quad \forall\, i, j \mid i \neq j.$$

Proposição 2.3.1. *Se a matriz A é definida positiva, e os vetores não-nulos $d_0, \ldots, d_k \in \Re^n$ são A-conjugados, então esses vetores são linearmente independentes [9].*

Demonstração. Provemos por contradição. Se os vetores não-nulos $d_0, \ldots, d_k$ são linearmente dependentes, então existem escalares σ_i nem todos nulos tal que $\sigma_0 d_0 + \sigma_1 d_1 + \cdots + \sigma_k d_k = 0$. Multiplicando por $d_i^T A \ \forall\ i = 0, \ldots, k$ temos:

$$\sigma_i d_i^T A d_i = 0$$

Uma contradição, uma vez que com A é definida positiva, temos que $d_i^T A d_i > 0$. ■

As direções d_k são obtidas pelo processo de Gram-Schmidt aplicado ao vetor $-g_k$

e nas direções anteriores $d_0, \ldots, d_{k-1}$ [16]. É possível acompanhar este procedimento visualmente, ou seja, iniciando com dois vetores v_1 e v_2 e assim estender para $k-$vetores. Fazemos $u_1 = v_1$, daí $u_2 = v_2 - proj_{u_1}(v_2)$. Para k, $u_k = v_k - \sum_{j=1}^{k-1} proj_{u_j}(v_k)$. Como mostra a Figura 2.1,

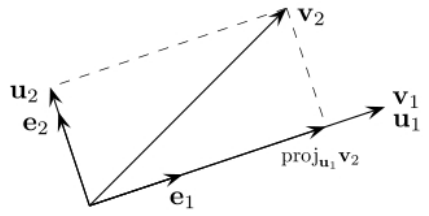

Figura 2.1: Ortogonalização dos vetores v_1 e v_2

Aplicando esse processo, obtemos a fórmula de d_k

$$d_k = -g_k + \sum_{j=0}^{k-1} \frac{g_k^T A d_j}{d_j^T A d_j} d_j,$$

e assim encontrarmos a direção procurada. Em se tratando de funções quadráticas, podemos fazer uso da fórmula explícita do seu gradiente. Assim, é possível obter a fórmula para d_k utilizando outro procedimento conforme exibimos a seguir.

Consideremos x_k e x_{k+1} os minimizadores respectivamente de $f(x)$ nas iterações k, $k+1$ e s_k o deslocamento feito numa iteração k, de x_k até x_{k+1}, ou seja,

$$s_k = x_{k+1} - x_k.$$

Como x_{k+1} é o minimizador da função na iteração $k+1$,

$$\min f(x) \quad \text{s.a} \quad x \in \text{span}\{\, g_0, \ldots, g_k \,\}$$

e o ponto x_k é o minimizador da função na iteração k,

$$\min f(x) \quad \text{s.a} \quad x \in \text{span}\{\, g_0, \ldots, g_{k-1} \,\}$$

então,

$$s_k \in span\{g_0, \ldots, g_k\}.$$

Logo, podemos escrever s_k como uma combinação linear dos vetores que geram o espaço

$$\begin{aligned} s_k &= \sum_{i=0}^{k} \sigma_i g_i \\ &= \sigma_0 g_0 + \sigma_1 g_1 + \cdots + \sigma_k g_k \\ &= \sigma_0 s_0 + \sigma_1 s_1 + \cdots + \sigma_{k-1} s_{k-1} - \alpha g_k. \end{aligned}$$

Dividindo a igualdade anterior por α e definindo $\frac{s_k}{\alpha} = d_k$ temos,

$$\begin{aligned} d_k = \frac{s_k}{\alpha} &= \frac{\sigma_0}{\alpha} s_0 + \frac{\sigma_1}{\alpha} s_1 + \cdots + \frac{\sigma_{k-1}}{\alpha} s_{k-1} - g_k \\ &= w_0 s_0 + w_1 s_1 + \cdots + w_{k-1} s_{k-1} - g_k. \end{aligned}$$

Quando multiplicamos a igualdade anterior por $s_j^T A, \ \forall \ j = 0, \cdots, k-1$, encontramos a expressão dos $w's$. Logo,

$$\begin{aligned} d_k &= -g_k + w_0 s_0 + w_1 s_1 + \cdots + w_{k-1} s_{k-1} \\ s_j^T A d_k &= -s_j^T A g_k + w_0 s_j^T A s_0 + w_1 s_j^T A s_1 + \cdots + w_j s_j^T A s_j + \cdots + w_{k-1} s_j^T A s_{k-1} \\ s_j^T A \frac{s_k}{\alpha_k} &= -s_j^T A g_k + w_0 s_j^T A s_0 + w_1 s_j^T A s_1 + \cdots + w_j s_j^T A s_j + \cdots + w_{k-1} s_j^T A s_{k-1} \end{aligned}$$

Mas, pela Definição (2.1) temos

$$0 = -s_j^T A g_k + w_j s_j^T A s_j$$

$$w_j = \frac{s_j^T A g_k}{s_j^T A s_j} \quad ; \quad j = 0, \ldots, k-1.$$

Observação 2.2. –

Fazendo a diferença de g_{j+1} e g_j encontramos,

$As_j = g_{j+1} - g_j$

Quando multiplicamos por g_k à esquerda temos

$$\begin{aligned} g_k^T A s_j &= g_k^T g_{j+1} - g_k^T g_j \\ \left(g_k^T A s_j\right)^T &= 0 \quad para\ j = 0, \ldots, k-2 \\ s_j^T A g_k &= 0 \quad para\ j = 0, \ldots, k-2 \end{aligned}$$

$\therefore \quad w_0 = \cdots = w_{k-2} = 0$

Agora substituindo a expressão de $w_0, \ldots, w_{k-1}$ em (2.3)

$$\begin{aligned} d_k &= -g_k + w_{k-1} s_{k-1} \\ &= -g_k + \frac{s_{k-1}^T A g_k}{s_{k-1}^T A s_{k-1}} s_{k-1} \end{aligned}$$

Como A é simétrica e $As_{k-1} = g_k - g_{k-1}$

$$\begin{aligned} d_k &= -g_k + \frac{g_k^T [g_k - g_{k-1}]}{s_{k-1}^T [g_k - g_{k-1}]} s_{k-1} \\ &= -g_k + \frac{g_k^T g_k}{-s_{k-1}^T g_{k-1}} s_{k-1} \\ &= -g_k - \frac{g_k^T g_k}{\alpha_{k-1} d_{k-1}^T g_{k-1}} \alpha_{k-1} d_{k-1} \end{aligned}$$

$$\begin{aligned} &= -g_k - \frac{g_k^T g_k}{d_{k-1}^T g_{k-1}} d_{k-1} \\ &= -g_k - \frac{g_k^T g_k}{[-g_{k-1} + w_{k-2} s_{k-2}]^T g_{k-1}} d_{k-1} \\ &= -g_k + \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}} d_{k-1} \end{aligned} \tag{2.3}$$

Concluímos então que,

$$d_k = -g_k + \beta_k d_{k-1},$$

onde,

$$\beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}.$$

Desta maneira, temos a seguinte proposição.

Proposição 2.3.2. *As direções do Método dos Gradientes Conjugados são geradas por*

$$d_0 = -g_0;$$

$$d_k = -g_k + \beta_k d_{k-1}, \quad k = 1, \ldots, n-1,$$

em que β_k é dado por

$$\beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}.$$

O algoritmo do método dos Gradientes Conjugados encontra-se a seguir.

Algoritmo 1 Gradientes Conjugados - Quadrática

Dados x_0, ϵ. Faça $k = 0$

Passo 1

$Calcule\ g_k$

se $g_k < \epsilon$ **então**

$Pare!$

fim se

Passo 2

se $k \geq 1$ **então**

$Calcule$

$\beta_k = \frac{g_k^T g_k}{g_{k-1}^T g_{k-1}}$

fim se

Passo 3

se $k = 0$ **então**

$d_k = -g_k$

senão

$d_k = -g_k + \beta_k d_{k-1}$

fim se

Passo 4

$Calcule$

$\alpha_k = \frac{g_k^T g_k}{d_k^T A d_k}$

Passo 5

$Calcule$

$x_{k+1} = x_k + \alpha_k d_k$

Passo 6

$Faça$ ***k=k+1*** $e\ volte\ ao$ ***passo 1***

2.3.2 Análise de Convergência do Método

Teorema 2.3. *O Método dos Gradientes Conjugados aplicado à minimização de quadráticas encontra uma solução do sistema linear $Ax = 0$ ou calcula uma direção ao longo do qual a quadrática tende a $-\infty$, em no máximo n passos, sendo n o número de autovalores distintos de A [5, 9].*

Demonstração. Pelo fato de A ser definida positiva, o problema (2.1) tem um único ponto estacionário $x^* \in \Re^n$. Pela convexidade de f, este ponto é a solução global do problema. Pela Proposição (2.3.1), o conjunto de vetores $d_0, d_1, \ldots, d_{n-1}$ forma uma base do $\Re^n$. Portanto, podemos representar o vetor $x^* - x_0$ como

$$\begin{aligned} x^* - x_0 &= \sigma_0 d_0 + \sigma_1 d_1 + \cdots + \sigma_{n-1} d_{n-1} \\ &= \sum_{i=0}^{n-1} \sigma_i d_i \end{aligned}$$

Agora multiplicando a equação anterior por $d_k^T A,\ 0 \leq k < n$, obtemos

$$d_k^T A(x^* - x_0) = \sigma_k d_k^T A d_k,$$

em que os termos $d_k^T A d_i = 0,\ k \neq i$, pela propriedade de A-conjugados. Assim,

$$\sigma_k = \frac{d_k^T A(x^* - x_0)}{d_k^T A d_k}.$$

Agora podemos escrever,

$$x_k = x_0 + \alpha_0 d_0 + \alpha_1 d_1 + \cdots + \alpha_{k-1} d_{k-1}$$

Portanto

$$x_k - x_0 = \alpha_0 d_0 + \alpha_1 d_1 + \cdots + \alpha_{k-1} d_{k-1}$$

Então escrevendo,

$$x^* - x_0 = (x^* - x_k) + (x_k - x_0)$$

e multiplicando por $d_k^T A$, nós obtemos

$$d_k^T A(x^* - x_0) = d_k^T A(x^* - x_k) = -d_k^T g_k$$

uma vez que $g_k = Ax_k - b$ e $Ax^* = b$. Deste modo,

$$\sigma_k = \frac{d_k^T g_k}{d_k^T A d_k} = \alpha_k$$

e $x^* = x_n$, que completa a prova. ■

A proposição a seguir diz a repeito da taxa de convergência do método quando aplicado em funções quadráticas.

Proposição 2.3.3. *Suponha que A tenha $n - r$ autovalores num intervalo [a, b] com $a > 0$, e que os autovalores r restantes sejam maiores do que b. Então para cada x_0, o vetor x_{k+1} gerado após $k + 1$ passos do método dos Gradiente Conjugados satisfaz,*

$$f(x_{k+1}) \leq \left(\frac{b-a}{b+a}\right)^2 f(x_k).$$

Demonstração. A demonstração desta proposição pode ser vista em Bertsekas [16]. ■

Assim como no método de Máxima Descida, a taxa de convergência do método dos Gradientes Conjugados é delimitada pela mesma fórmula. E como consideramos uma quadrática, a taxa de convergência está na relação entre os comprimentos dos eixos dos contornos elípticos de $f(x)$, ou seja, a excentricidade do elipsóide [7]. Consequentemente as propriedades de $f(x)$ aqui também são consideradas: quanto maior a excentricidade do eplisóide maior será a lentidão da convergência, ou seja, quanto maior a distância dos

autovalores mais lenta a convergência. O número de iterações para a convergência do método dos Gradientes Conjugados é proporcional à raiz do número espectral $\rho(A) = \frac{b}{a}$, em que b é o maior e a o menor dos autovalores de A e é chamado de número de condicionamento da matriz [15]. Deste modo, a taxa de convergência é estabelecida pela seguinte fórmula [16]

$$\left(\frac{b-a}{b+a}\right)^2 = \left(\frac{\rho(A)-1}{\rho(A)+1}\right)^2.$$

Capítulo 3

Métodos dos Gradientes Conjugados Não Linear

O método dos Gradiente Conjugados foi inicialmente proposto para resolução de problemas quadráticos, diretamente relacionado ao cálculo de sistemas lineares. Entretanto, é possível utilizá-lo na minimização de funções não quadráticas. Fletcher-Reeves [2] estabaleceram uma primeira modificação para conseguir desempenhar este papel em tais problemas, adaptando o método para não somente minimizar funções quadráticas, mas também funções não-lineares de qualquer natureza, desde que a função f seja contínua e difereciável, de modo a assegurar que o gradiente ∇f possa ser calculado (permitindo assim o seu uso para a resolução de um conjunto muito maior de problemas, oriundos de engenharia, redes neurais e regressão linear [15], por exemplo).

O algoritmo para o caso de problemas com funções não quadráticas tem estrutura similar para o caso envolvendo funções quadráticas porém, com algumas ressalvas. O algoritmo para estas funções firma sua base no Algoritmo 1 mas, a busca linear aqui não é mais exata, ou seja, para problemas não quadráticos a atuação da busca linear ao longo da busca da direção é mais complicado do que no caso de quadráticas, assim

deve-se estabelecer uma busca linear em que a identificação de um ponto se aproxime do mínimo de $f(x)$ ao longo da direção d_k e tenta sempre garantir que o gradiente $\nabla f(x)$ seja ortogonal à direção buscada [15].

Fletcher e Reeves mostraram que para estender o método dos Gradientes Conjugados para funções da forma do problema (3.1), deveríamos fazer duas mudanças no Algoritmo 1. A primeira delas relata que é preciso identificar um outro tamanho de passo α que também minimize f ao longo da direção d_k que encontramos com a busca linear exata (ou que pelo menos garanta um decréscimo suficiente da imagem). A segunda importante observação é que para o caso de funções f gerais, o gradiente não necessariamente pode ser representado como um resíduo de um sistema linear.

O método, quando aplicado a um problema não linear irrestrito

$$\text{minimizar } \{f(x) \mid x \in \Re^n\} \tag{3.1}$$

em que $f : \Re^n \mapsto \Re$ é uma função contínua, diferenciável e limitada inferiormente, gera uma sequência $\{x_k\}$, iniciado pelo ponto $x_0 \in \Re^n$, da seguinte forma,

$$x_{k+1} = x_k + \alpha_k d_k, \tag{3.2}$$

onde α_k é o tamanho do passo obtido pela busca linear e d_k são as direções encontradas por

$$d_{k+1} = -\nabla f(x_{k+1}) + \beta_k d_k. \tag{3.3}$$

3.1 Busca Linear

Na otimização de problemas não quadráticos, a escolha do parâmetro da busca linear α, deve ser feita de forma criteriosa, pois a direção d_k pode não ser uma direção de descida [8] e o cálculo deste parâmetro pode custar muito do algoritmo quando estamos nos referindo ao tempo e por ser muito onerosa a busca linear exata não é utilizada. Determinamos o tamanho do passo α do seguinte modo:

$$\alpha = arg \overset{\alpha>0}{min} f(x_k + \alpha d_k).$$

Definimos uma função unidimensional e busca-se o mínimo global dessa função.

$$\phi(\alpha) = f(x_k + \alpha d_k), \quad \alpha > 0. \tag{3.4}$$

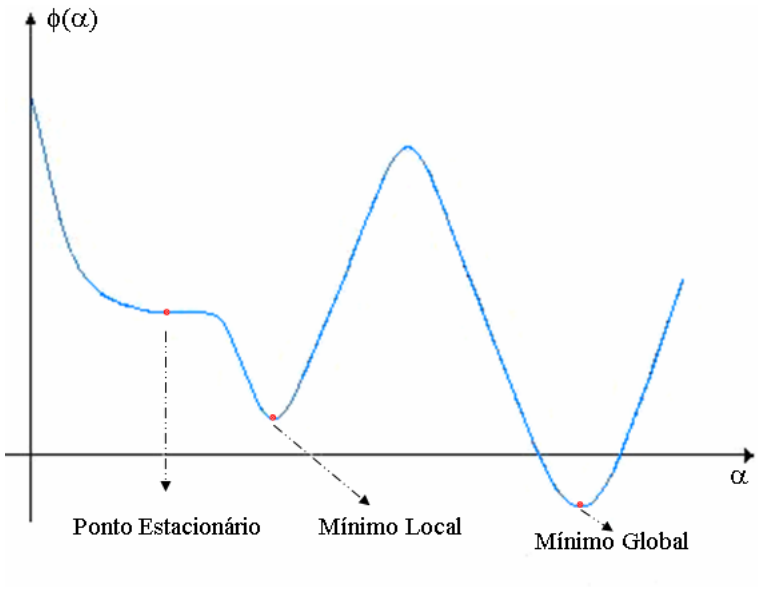

Figura 3.1: Tamanho ideal do passo α, dado pelo mínimo global.

Exigir um tamanho de passo α que apenas faz com que haja uma redução da imagem

da função objetivo a cada iteração pode não ser um bom "negócio", ou seja, não garantir uma convergência rápida,

$$f(x_{k+1}) < f(x_k).$$

Para ilustrarmos esta situação podemos tomar como exemplo a função $f(x) = x^2$, com $x_0 = 2$. A sequência $((-1)^k(1 + 2^{-k}))$ é tal que $f(x_{k+1}) < f(x_k)$. Porém $\lim_{k\to\infty} f(x_k) = 1$, enquanto que o minimizador de f é 0. A Figura 3.2 enfatiza que mesmo decrescendo a imagem da função não garante convergência rápida.

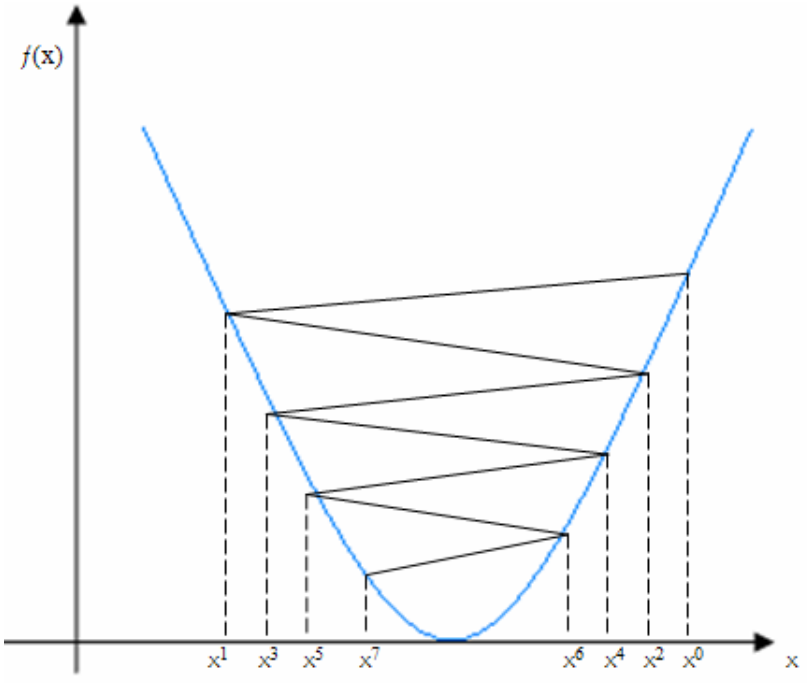

Figura 3.2: Decréscimo de f não garante convergência (rápida).

Para contornar esse comportamento devemos tomar condições mais exigentes, conforme a seguir.

3.1.1 Condições de Wolfe

A primeira condição impõe que o tamanho do passo α deve satisfazer a seguinte desigualdade:

$$f(x_k + \alpha_k d_k) \quad \leq \quad f(x_k) + c_1 \alpha_k \nabla f(x_k)^T d_k \quad : \quad r(\alpha) \tag{3.5}$$

Chamamos esta desigualdade de *Condição de Armijo* e ela estabelece que a diminuição $f(x_{k+1}) - f(x_k)$ é proporcional tanto ao tamanho do passo α quanto ao produto $\nabla f(x_k)^T d_k$.

A inclinação da reta $r(\alpha) = f(x_k) + c_1 \alpha_k \nabla f(x_k)^T d_k$ é negativa, pois $\nabla f(x_k)^T d_k < 0$. Esta condição é satisfeita para valores de $\alpha > 0$ pequenos e nos testes computacionais feitos (que serão exibidos nos próximos capítulos), assumimos $c_1 = 0,0001$.

A Figura 3.3 ilustra a ideia da *Condição de Armijo*.

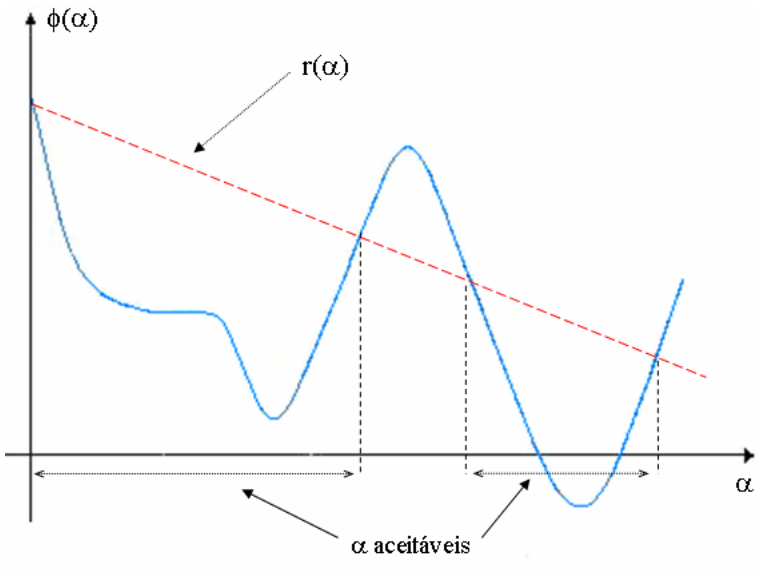

Figura 3.3: Condição Armijo.

Se exigirmos que o tamanho do passo α apenas satisfaça a condição de Armijo o método não consegue ter um bom desempenho, pois se adotamos somente esta condição as

escolhas dos passos são muito pequenos e mais uma vez nos depararíamos com uma convergência lenta. Para assegurar um melhor desempenho introduzimos mais uma condição no algoritmo da Busca Linear que chamamos de *Condição de Curvatura* e que definimos como:

$$\nabla f(x_k + \alpha_k d_k)^T d_k \geq c_2 \nabla f(x_k)^T d_k \tag{3.6}$$

onde $0 < c_1 < c_2 < 1$. O lado esquerdo da condição 3.6 é exatamente a derivada de $\phi'(\alpha)$, é ela garante que a inclinação de $\phi(\alpha)$ é maior que c_2 vezes a inclinação inicial $\phi'(0)$. A condição de curvatura é importante pois se a inclinação $\phi'(\alpha)$ é muito negativa isto significa que ainda é possível reduzir f percorrendo a direção d_k. E em caso contrário, o algoritmo pára.

A Figura 3.4 exibe didaticamente a ideia da segunda condição Wolfe.

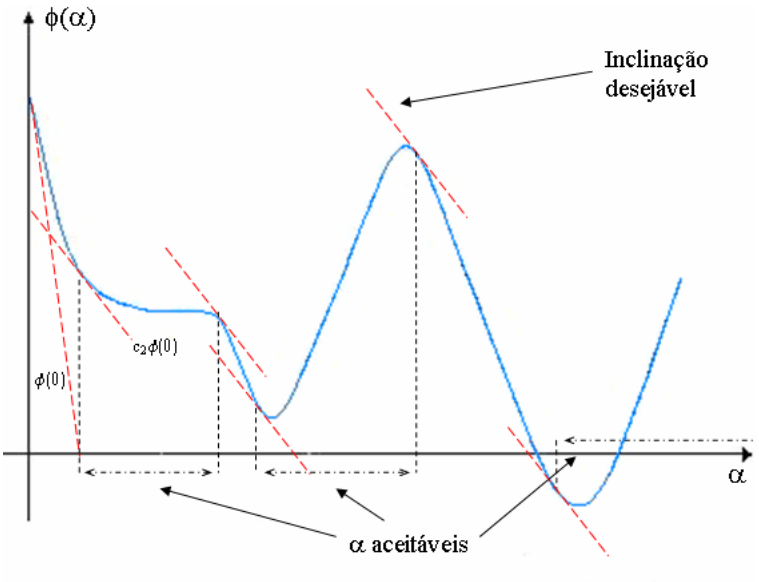

Figura 3.4: Condição de Curvatura.

Nos testes numéricos assumimos $c_2 = 0,1$. A Figura 3.5 exibe a Condição de Armijo e Curvatura juntas e podemos ver que há alguns tamanhos de passos α que satifazem

tanto a condição de Armijo quanto a condição de Curvatura mas, tais passos α mesmo satisfazendo essas condições estão afastados do mínimo de ϕ.

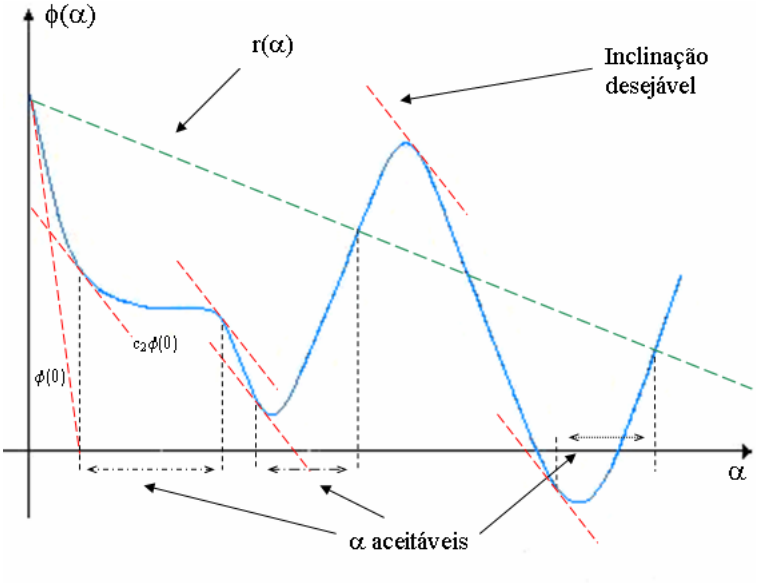

Figura 3.5: Condições de Wolfe.

Para contornar este possível problema, a Condição de Curvatura pode sofrer uma pequena modificação de forma a escolher aqueles α's que estão próximos do minimizador global ou ao menos de um ponto crítico de ϕ e assim eliminamos os pontos que estão mais afastados do ponto crítico reduzindo inclusive o número de iterações [8].

Ao fazermos esta modificação na condição de Curvatura chamamos esta intervenção unida com a condição de Armijo de *Condição Forte de Wolfe* e é como segue.

$$f(x_k + \alpha_k d_k) \leq f(x_k) + c_1 \alpha_k \nabla f(x_k)^T d_k \tag{3.7}$$

$$|\nabla f(x_{k+1})^T d_k| \leq |c_2 \nabla f(x_k)^T d_k| \tag{3.8}$$

onde $0 < c_1 < c_2 < \frac{1}{2}$. A Figura 3.6 mostra a ideia da condição forte de Wolfe.

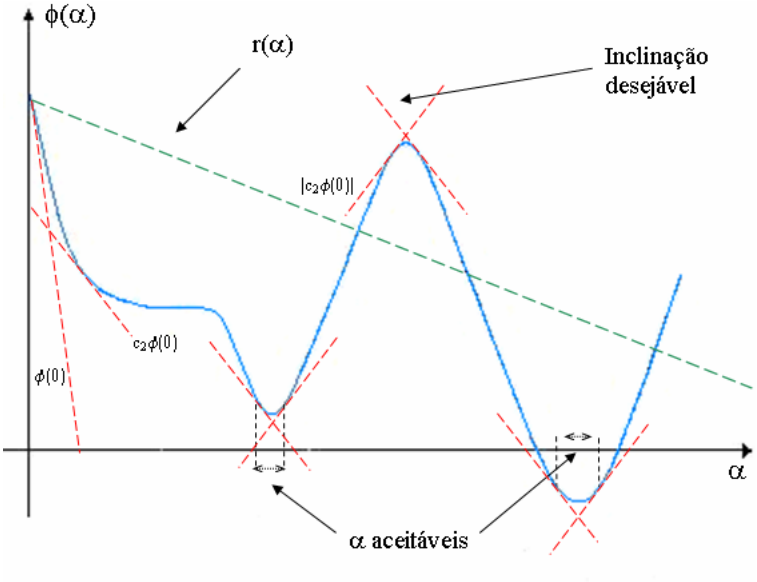

Figura 3.6: Condições Forte de Wolfe.

Quando fazendo o produto interno de (3.3) pelo vetor gradiente $\nabla f(x)$

$$\nabla f(x_{k+1})^T d_{k+1} = - \ \nabla f(x_{k+1})\ ^2 + \beta_k \nabla f(x_{k+1})^T d_k. \tag{3.9}$$

e fazendo uso das Condições de Wolfe (3.5) e (3.6) temos,

$$\nabla f(x_{k+1})^T d_{k+1} = - \ \nabla f(x_{k+1})\ ^2 + \beta_k \nabla f(x_{k+1})^T d_k < 0. \tag{3.10}$$

Afirmamos [8] que qualquer procedimento de busca linear [1] que satisfaça as condições (3.5) e (3.6), garante que todas as direções d_k são de descida para a função f.

3.1.2 Algoritmo - Busca Linear

A busca pelo comprimento do passo α é feito pela Busca Linear e exige também um pouco do algoritmo do Método dos Gradientes Conjugados, por isso deve ser um

[1] Se a busca linear α é exata temos que $\nabla f(x_{k+1}) d_k < 0$, assim em (3.9), d_k é uma direção de descida.

procedimento que realize a busca pelo α reduzindo a função objetivo f, porém que não gaste muito tempo para encontrá-lo. Nesta seção discutimos alguns algoritmos de Busca Linear que encontram um tamanho de passo α "ideal".

Busca Linear Exata

Devemos ter uma Busca Linear que encontre o melhor tamanho de passo α possível numa determinada direção d_k. Para os problemas em geral (não quadráticos) encontrar analiticamente uma busca linear exata que nos dê um tamanho de passo "ideal" é muito difícil e por isso utilizamos para esses problemas outras buscas lineares que mencionamos a seguir, mas para os problemas quadráticos a busca linear exata é extremamente simples de se obter e ela já foi explorada no capítulo anterior. O tamanho do passo α na busca linear exata tem a seguinte expressão:

$$\alpha = \frac{-\nabla f(x_k)^T d_k}{d_k^T A d_k}. \tag{3.11}$$

Busca Linear Backtracking

A busca linear Backtracking tem como estratégia começar com um tamanho de passo α arbitrário e ir decrescendo sucessivamente até que α reduza a função objetivo f.

Algoritmo 2 Busca Linear Backtracking

Dados $\alpha = 1$; $iter = 0$ e $itermax$ =número máximo de iteração
enquanto $iter < itermax$ **faça**
  $iter \leftarrow iter + 1$;
  $\overline{x} \leftarrow x_k + \alpha d_k$;
  se $f(\overline{x}) < f(x_k)$ **então**
    $PARE$!
  senão
    $\alpha \leftarrow \alpha/2$;
  fim se
fim enquanto

Busca Linear de Armijo

Podemos fazer uma modificação no algoritmo da busca linear de Armijo para que tenha como estratégia encontrar o primeiro tamanho de passo α não muito grande e, em seguida, que também não seja muito pequeno. Conforme definimos em (3.1) assim,
O tamanho do passo é considerado não muito grande, se

$$\phi(\alpha_k) \leq \phi(0) + c_1\alpha\phi'(0). \tag{3.12}$$

E considerado não muito pequeno, se

$$\phi(\alpha_k\eta) > \phi(0) + c_1\alpha\eta\phi'(0). \tag{3.13}$$

Algoritmo 3 Busca Linear Armijo

Dados $\alpha \leftarrow 1$; $c_1 \leftarrow 0,2$ e $\eta \leftarrow 2$
se $\phi(\alpha_k) \leq \phi(0) + c_1\alpha\phi'(0)$ **então**
  enquanto $\phi(\alpha_k) \leq \phi(0) + c_1\alpha\phi'(0)$ **faça**
    $\alpha_k \leftarrow \eta\alpha_k$;
  fim enquanto
senão
  enquanto $\phi(\alpha_k) > \phi(0) + c_1\alpha\phi'(0)$ **faça**
    $\alpha_k \leftarrow \alpha_k/\eta$;
  fim enquanto
fim se

Busca Linear de Wolfe

Mencionamos também o algoritmo de busca linear que encontra um tamanho de passo α satisfazendo as condições de Wolfe, ou seja,

$$\phi(\alpha_k) \leq \phi(0) + c_1\alpha\phi'(0), \tag{3.14}$$

$$\phi'(\alpha_k) \geq c_2\phi'(0). \tag{3.15}$$

Definimos também $g(\alpha_k) = \phi'(\alpha_k) - c_1\phi'(0)$, e $g'(\alpha_k) = \phi(\alpha_k) - \phi(0) - c_1\alpha\phi'(0)$. O Algoritmo 4 da busca linear de Wolfe tem duas etapas, a primeira encontra um intervalo $[a, b]$ que satisfaz as condiçoes (3.7) e (3.8) e a segunda etapa realiza a redução de f no intervalo $[a, b]$.

Algoritmo 4 Busca Linear Wolfe

Dados $a \leftarrow 0$, c_1, c_2 e ϵ
Etapa 1 : Encontra um intervalo a, b valido
$\alpha_k \leftarrow -g'(0)/g''(0)$; $\Delta_1 \leftarrow 0$; $\Delta_2 \leftarrow 0$;
enquanto $(\Delta_1 = 0)$ **faça**
  se $(g'(\alpha_k) < -\epsilon)$ **então**
    $a \leftarrow \alpha_k$;
    $\alpha_k \leftarrow 2\alpha_k$;
  senão se $(g'(\alpha_k) > \epsilon)$ **então**
    $b \leftarrow \alpha_k$;
    $\Delta_1 \leftarrow 1$;
  senão
    $\Delta_1 \leftarrow 1$;
    $\Delta_2 \leftarrow 1$;
  fim se
fim enquanto
  Etapa 2 : Encontra $\bar{\alpha_k}$ no intervalo a, b.
enquanto $(\Delta_2 = 0)$ **faça**
  se $(g'(\alpha_k) < -\epsilon)$ **então**
    $a \leftarrow \alpha_k$;
    $\alpha_k \leftarrow a + \frac{1}{2}(b - a)$;
  senão se $(g'(\alpha_k) > \epsilon)$ **então**
    $b \leftarrow \alpha_k$
    $\alpha_k \leftarrow a + \frac{1}{2}(b - a)$;
  senão
    $\Delta_2 \leftarrow 1$;
  fim se
fim enquanto
  $\bar{\alpha_k} \leftarrow \alpha_k$

3.2 As Direções d_k

Diferentemente dos problemas quadráticos, as várias expressões de β_k para problemas não quadráticos não são equivalentes e pesquisadores buscam a melhor escolha para determinados problemas [15]. Existem casos em que a função f tem vários mínimos locais e o método dos Gradientes Conjugados pode não convergir para o mínimo global, ou sequer pode encontrar um mínimo local quando f não é limitada inferiormente [15]. As fórmulas mais conhecidas (ver [17, 7, 16, 5, 9, 8, 15, 14, 18]) para β_k são as de Hestenes-Stiefel(HS) [1], Fletcher-Reeves(FR) [2] e Polak-Ribière(PR) [6], a saber:

- Hestenes-Stiefel(HS)

$$\beta_{k+1} = \frac{\nabla f(x_{k+1})^T [\nabla f(x_{k+1}) - \nabla f(x_k)]}{d_k^T [\nabla f(x_{k+1}) - \nabla f(x_k)]}; \tag{3.16}$$

- Fletcher-Reeves(FR)

$$\beta_{k+1} = \frac{\nabla f(x_{k+1})^T \nabla f(x_{k+1})}{\nabla f(x_k)^T \nabla f(x_k)}; \tag{3.17}$$

- Polak-Ribière(PR)

$$\beta_{k+1} = \frac{\nabla f(x_{k+1})^T [\nabla f(x_{k+1}) - \nabla f(x_k)]}{\nabla f(x_k)^T \nabla f(x_k)}. \tag{3.18}$$

Desta maneira, a classe do método dos Gradientes Conjugados Não Linear se diferencia, em geral, pelas atualizações para β_k. Na próxima seção descreveremos de maneira sucinta alguns dos mais estudados β_k da literatura, ressaltando os detalhes primordiais de sua obtenção.

3.2.1 As Variações do Escalar β_k

O desenvolvimento e a escolha do parâmetro de atualização β que não utilizam a manipulação da hessiana têm maior preferência[2] sobre os β que utilizam a hessiana a cada iteração [14].

Dizemos anteriormente que quando estamos trabalhando com funções quadráticas a escolha de β é única, mas quando trabalhamos com funções não-quadráticas cada escolha leva à desempenhos diferentes [14].

A primeira variante de β que expomos é de Fletcher-Reeves e como já foi dito, sua menção é datada de 1964 e focava diretamente na otimização de funções não-lineares diferentemente de Hestenes-Stiefel[3] que evidenciava sistemas lineares em que a matriz A fosse simétrica definida positiva.

A fórmula do parâmetro de atualização β de Fletcher-Reeves é expressa como segue:

$$\beta_{k+1}^{FR} = \frac{\nabla f(x_{k+1})^T \nabla f(x_{k+1})}{\nabla f(x_k)^T \nabla f(x_k)}.$$

Seguindo uma ordem cronológica de criação das variações de β, o próximo β que citamos é de Polak & Ribière [14].

Quando se trata de funções quadráticas e com busca linear α exata, o β tanto de Fletcher-Reeves quanto de Polak-Ribière se comportam de maneira idêntica. Entretanto, quando procedemos com funções não-lineares e sobretudo com busca linear inexata, seus algoritmos diferem um do outro [8]. Com o β de Fletcher-Reeves o algoritmo converge se o chute inicial é suficientemente próximo do mínimo desejado (mais eficiente para a busca por mínimos locais). No entanto, o β de Polak-Ribière, quando converge, o faz muito mais rapidamente, ou seja, Polak-Ribière na prática tende a ser mais eficiente entre os dois [15]

[2] Principalmente, em problemas de grande porte.

[3] Embora o β proposto por Hestenes-Stiefel seja tão utilizado na literatura quanto o proposto por Fletcher-Reeves e o proposto por Polak-Ribière, este não será considerado na análise proposta nesta dissertação pois sua estrutura é específica para problemas quadráticos

(existem casos raros citados na literatura em que este pode ficar ciclando, sem convergir).

Outro fato surpreendente sobre o β de Polak-Ribière é que as condições forte de Wolf (3.5) e (3.6) não garantem que d_k seja sempre uma direção de descida. Quando estratégias adicionais são implementadas com o intuito de garantir que $\beta_k \geq 0$, então uma simples adaptação das condições forte de Wolfe garante que a propriedade de descida se mantém [8].

A fórmula do parâmetro de atualização β de Polak-Ribière é expressa como segue:

$$\beta_{k+1}^{PR} = \frac{\nabla f(x_{k+1})^T [\nabla f(x_{k+1}) - \nabla f(x_k)]}{\nabla f(x_k)^T \nabla f(x_k)}.$$

3.2.2 Algoritmo - Método dos Gradientes Conjugados

Discutimos nas seções anteriores como obtemos os cálculos das variáveis direções d_k e dos passos α_k para então incrementarmos x_k encontrando o novo ponto x_{k+1}. Segue o algoritmo do Método dos Gradientes Conjugados para funções gerais (não quadráticas).

Algoritmo 5 Método dos Gradientes Conjugados - Não Quadrático

Dados $k \leftarrow 0$; $x_k \leftarrow$ *Chute inicial* e $g_k \leftarrow \nabla f(x_k)$

enquanto $(||g_k||_2^2 \geq \epsilon)$ ou $(k < itermax)$ **faça**
  se $(k > 0)$ **então**
    $\beta_{k+1} \leftarrow$ *Escolhe* β : (3.17), (3.18) *ou* (5.5);
    $d_{k+1} \leftarrow -g_{k+1} + \beta_{k+1} d_k$;
  senão
    $d_{k+1} \leftarrow -g_k$;
  fim se
  $\alpha_{k+1} \leftarrow$ *Busca Linear* : (2), (3) *ou* (4);
  $x_{k+1} \leftarrow x_k + \alpha_{k+1} d_{k+1}$;
  $g_{k+1} \leftarrow \nabla f(x_{k+1})$;
  $k \leftarrow k + 1$.
fim enquanto

3.3 Análise de Convergência

Vamos apresentar a convergência do Método dos Gradientes Conjugados aplicado a problemas não lineares.

Assumimos primeiramente que todas as direções d_k satisfazem a condição de descida,

$$g_k^T d_k < 0, \quad \forall\ k \geq 1. \tag{3.19}$$

Como hipóteses, adotamos,

Hipótese 1 - H1: A função objetivo $f(x)$ é limitada inferiormente no $\Re^n$ e é continuamente diferenciável.

Hipótese 2 - H2: O gradiente $g(x)$ da função objetivo $f(x)$ é Lipschitz contínuo no conjunto B convexo aberto que contém $L(x_0) = \{x \mid f(x) \leq f(x_0)\}$, isto é, existe um $L > 0$ que,

$$g(x) - g(y) \leq L \ \ x - y \ , \forall x, y \in B. \tag{3.20}$$

O tamanho do passo α_k em (3.2) é calculado através da realização de uma busca linear. Para provar a convergência global usa-se as condições forte de Wolfe, requerendo que α_k satisfaça (3.5) e (3.6).

O seguinte resultado muito importante foi obtido por Zoutendijk [19] e Wolfe [20] - [21].

Lema 3.1. *Suponhamos (H1) e (H2) verdadeiras. Considere qualquer método de iteração da forma (3.2)-(3.3), em que d_k satisfaz (3.19) e α_k é obtido pela busca linear de Wolfe (condições de Armijo e de Curvatura). Então,*

$$\sum_{k=1}^{\infty} \frac{(g_k^T d_k)^2}{||d_k||^2} < +\infty. \tag{3.21}$$

O seguinte Teorema é um resultado geral para o Método dos Gradiente Conjugados com a busca linear de Wolfe (Armijo).

Teorema 3.1. *Suponhamos (H1) e (H2) verdadeiras. Considere qualquer método de iteração da forma (3.2)-(3.3) com d_k satisfazendo (3.19) e com as condições fortes de Wolfe (3.5) e (3.6). Então:*

$$\lim_{k\to\infty} inf\,||g_k|| = 0, \tag{3.22}$$

ou

$$\sum_{k=1}^{\infty} \frac{||g_k||^4}{||d_k||^2} < +\infty. \tag{3.23}$$

Demonstração. De (3.3), temos que para todos os $k \geq 2$,

$$d_k + g_k = \beta_k d_{k-1}. \tag{3.24}$$

Aplicando o quadrado da norma-2 em (3.24), obtemos

$$||d_k||^2 = -||g_k||^2 - 2g_k^T d_k + \beta_k^2 ||d_{k-1}||^2. \tag{3.25}$$

Decorre dessa relação e de (3.19) que

$$||d_k||^2 \geq \beta_k^2 ||d_{k-1}||^2 - ||g_k||^2. \tag{3.26}$$

De (3.3) temos que

$$g_k^T d_k - \beta_k g_k^T d_{k-1} = -||g_k||^2, \tag{3.27}$$

que, com a condição de Wolfe (3.6),

$$|g_k^T d_k| + \sigma|\beta_k||g_{k-1}^T d_{k-1}| \geq ||g_k||^2. \tag{3.28}$$

A desigualdade anterior e pela desigualdade de Cauchy-Schwartz,

$$(g_k^T d_k)^2 + \beta_k^2(g_{k-1}^T d_{k-1}) \geq c_1||g_k||^4, \tag{3.29}$$

em que $c_1 = (1+\sigma^2)^{-1}$ é uma constante positiva. Portanto, segue de (3.28) e (3.29) que

$$\begin{aligned}
\frac{(g_k^T d_k)^2}{||d_k||^2} \quad + \quad & \frac{(g_{k-1}^T d_{k-1})^2}{||d_{k-1}||^2} = \frac{1}{||d_k||^2}\left[(g_k^T d_k)^2 + \frac{||d_k||^2}{||d_{k-1}||^2}(g_{k-1}^T d_{k-1})^2\right] \\
\geq \quad & \frac{1}{||d_k||^2}\left[(g_k^T d_k)^2 + \beta_k^2(g_{k-1}^T d_{k-1}) - \frac{(g_{k-1}^T d_{k-1})^2}{||d_{k-1}||^2}||g_k||^2\right] \\
\geq \quad & \frac{1}{||d_k||^2}\left[c_1||g_k||^4 - \frac{(g_{k-1}^T d_{k-1})^2}{||d_{k-1}||^2}||g_k||^2\right].
\end{aligned} \tag{3.30}$$

Se (3.22) não é verdadeiro, as relações (3.30) e (3.21) implicam que a seguinte desigualdade

$$\frac{(g_k^T d_k)^2}{||d_k||^2} \quad + \quad \frac{(g_{k-1}^T d_{k-1})^2}{||d_{k-1}||^2} \geq \frac{c_1||g_k||^4}{2||g_k||^2} \tag{3.31}$$

vale para todo k suficientemente grande. Assim, a desigualdade (3.23) segue de (3.31) e (3.21). ■

O seguinte resultado é um Corolário do teorema anterior.

Corolário 3.2. *Suponhamos (H1) e (H2) verdadeiras. Considere qualquer método de iteração da forma (3.2)-(3.3) com d_k satisfazendo (3.19) e com as condições fortes de Wolfe (3.5) e (3.6). Se*

$$\sum_{k=1}^{\infty} \frac{||g_k||^t}{||d_k||^2} = +\infty, \tag{3.32}$$

para qualquer $t \in [0, 4]$, o método converge com (3.22) verdadeira.

Demonstração. Se (3.22) não é verdadeira, segue pelo Teorema 3.1 que

$$\sum_{k=1}^{\infty} \frac{||g_k||^4}{||d_k||^2} < +\infty. \tag{3.33}$$

Mas $||g_k)||$ é limitado e $t \in [0, 4]$, logo é fácil ver que (3.33) contradiz (3.32). Isto mostra que o Corolário é verdadeiro. ■

3.3.1 Convergência Global

Nesta seção, estabelecemos alguns resultados de convergência global para o método dos Gradientes Conjugados com os β's de Fletcher-Reeves e de Polak-Ribière. Os resultados obtidos se baseiam no fato de que se a relação de convergência (3.22) não for verdadeira, podemos deduzir que $\sum_{k=1}^{\infty} \frac{||g_k||^2}{||d_k||^2} = +\infty$ ou $\sum_{k=1}^{\infty} \frac{1}{||d_k||^2} = +\infty$, que com o Corolário 3.2, implica que (3.22) seja verdadeira, dando uma contradição.

Primeiramente, consideramos o método dos Gradientes Conjugados com Fletcher-Reeves da forma (3.2)-(3.3), em que β_k seja um escalar qualquer que satisfaça

$$\sigma|\beta_k| \leq \bar{\sigma}\beta_k^{FR} \tag{3.34}$$

para todo $k \geq 2$, em que σ é um parâmetro definido em (3.6) e $\bar{\sigma} \in (0, 1/2]$ é uma constante.

Teorema 3.3. *Suponhamos (H1) e (H2) verdadeiras. Considere qualquer método de iteração da forma (3.2)-(3.3) com as condições fortes de Wolfe (3.5) e (3.6), em que β_k satisfaça (3.34) com $\bar{\sigma} \in (0, 1/2]$, e*

$$||g_k||^2 \sum_{j=2}^{k} \prod_{i=j}^{k} \left(\frac{\beta_i}{\beta_i^{FR}} \right)^2 \leq c_2 k, \tag{3.35}$$

para uma constante $c_2 > 0$. Então,

$$\lim_{k\to\infty} inf\, ||g_k|| = 0. \tag{3.36}$$

Demonstração. De (3.3), $\beta_k^{FR} = ||g_k||^2/||g_{k-1}||^2$, (3.6) e (3.34), deduzimos que

$$\begin{aligned}
\frac{-g_k^T d_k}{||g_k||^2} &= 1 - \beta_k \frac{-g_k^T d_{k-1}}{||g_k||^2} = 1 - \left(\frac{\beta_k}{\beta_k^{FR}}\right) \frac{-g_k^T d_{k-1}}{||g_{k-1}||^2} \\
&\leq 1 + |\frac{\beta_k}{\beta_k^{FR}}| \frac{-\sigma g_{k-1}^T d_{k-1}}{||g_{k-1}||^2} \\
&\leq 1 + \sigma \left(\frac{-g_{k-1}^T d_{k-1}}{||g_{k-1}||^2}\right) \\
&\leq \cdots \\
&\leq \sum_{j=0}^{k-2} \bar{\sigma}^j + \bar{\sigma}^{k-1} \left(\frac{-g_1^T d_1}{||g_1||^2}\right) = \frac{1-\bar{\sigma}^k}{1-\bar{\sigma}} < \frac{1}{1-\bar{\sigma}}.
\end{aligned} \tag{3.37}$$

Da mesma forma, temos que

$$\frac{-g_k^T d_k}{||g_k||^2} \geq 1 - \bar{\sigma} \frac{1-\bar{\sigma}^{k-1}}{1-\bar{\sigma}} > 0, \tag{3.38}$$

pois $\bar{\sigma} \leq 1/2$. Portanto, d_k é uma direção de descida.

Por outro lado, segue de (3.25) que

$$||d_k||^2 = -2g_k^T d_k + \beta_k^2 ||d_{k-1}||^2. \tag{3.39}$$

Usando (3.39) recursivamente e observando que $d_1 = -g_1$, temos que

$$\begin{aligned}
||d_k||^2 &\leq -2g_k^T d_k - 2\sum_{j=2}^{k} \prod_{i=j}^{k} \beta_i^2 g_{j-1}^T d_{j-1} \\
&= -2g_k^T d_k - 2||g_k||^4 \sum_{j=2}^{k} \prod_{i=j}^{k} \left(\frac{\beta_k}{\beta_k^{FR}}\right)^2 \left(\frac{-g_{j-1}^T d_{j-1}}{||g_{j-1}||^4}\right).
\end{aligned} \tag{3.40}$$

Se o teorema não é verdadeiro, (3.35) mantém e existe uma constante γ positiva tal que

$$||g_k|| \geq \gamma, \quad \forall k. \tag{3.41}$$

Assim, resulta da desigualdade acima, (3.37) e (3.40) que

$$\frac{||d_k||^2}{||g_k||^2} \leq \frac{2}{1-\bar{\sigma}} \left[1 + \frac{||g_k||^2}{\gamma^2} \sum_{j=2}^{k} \prod_{i=j}^{k} \left(\frac{\beta_k}{\beta_k^{FR}} \right)^2 \right]. \tag{3.42}$$

A relação acima e (3.35) implica que

$$\sum_{k=1}^{\infty} \frac{||g_k||^2}{||d_k||^2} = +\infty. \tag{3.43}$$

Este, com Corolário 3.2, implica que $lim\ inf_k\ ||g_k|| = 0$. Isso completa a nossa prova. ■

Capítulo 4

Problemas Testes de Otimização Irrestrita - β^{FR} e β^{PR}

4.1 Introdução

Neste capítulo realizamos os testes numéricos destinados a resolução de alguns problemas quadráticos e outros selecionados do conjunto de problemas testes de Moré-Burton-Kenneth [3]. Adotamos como método de resolução o Método dos Gradientes Conjugados descrito no Capítulo 3. Dos diversos β's descritos na literatura, consideramos os de Fletcher-Reeves (que abreviaremos para FR) e Polak-Ribière (abreviado para PR), descritos na Seção 3.2.1.

Para cada um dos problemas foram realizados testes numéricos com várias dimensões, sendo reportados neste capítulo os resultados obtidos com 3 valores de dimensão específicas, a saber: a menor dimensão para a qual o problema foi elaborado, como por exemplo, 2 ou 3 e outras duas dimensões, $n = 12$ e $n = 36$, fixas para todos os problemas, exceto das funções quadráticas que foram criadas para serem testadas com dimensões 2, 10, 12, 36 e 100.

Optamos pela implementação do Algoritmo 1 na linguagem de programação C, visto que sua estrutura é muito flexível, robusta e confiável. O código fonte de implementação pode ser analisado no Anexo que segue na dissertação.

Os problemas testes que consideramos têm sua forma definida como sendo o somatório do quadrado de funções não-lineares $f_1, \ldots, f_m$

$$f(x) = min \left\{ \sum_{j=1}^{m} f_j^2(x) : x \in \Re^n \right\},$$

e assim obtemos um problema de minimização irrestrita, em que $f_j : \Re^n \to \Re$ para $j = 1, \ldots, m$ com $m \geq n$ (ver [3]).

O gradiente das funções f a serem minimizadas para os problemas desta classe é expresso da seguinte maneira:

$$\nabla f(x) = 2 \sum_{i=1}^{m} f_i(x) \nabla f_i(x).$$

em que $\nabla f_i(x) = J(x)$ (matriz Jacobiana).

4.1.1 Critério de Parada

Adotamos como *Critério de Parada* para os algoritmos as condições que o norma-2 do gradiente da função f seja suficientemente pequeno ou que o módulo da diferença da imagem no ponto atual pela imagem da função no ponto anterior também seja tão pequeno quanto quieramos. Assim temos o seguinte,

Algoritmo 6 Critério de Parada

$Calcule\ ||\nabla f(x_{k+1})||\ e\ |f(x_{k+1}) - f(x_k)|$
se $(||\nabla f(x_{k+1})|| \leq \epsilon$ ou $|f(x_{k+1}) - f(x_k)| \leq \epsilon)$ **então**
$Pare!$
fim se

4.1.2 Discussões

- Moré [3] impõe que ao trabalharmos com as funções aqui testadas devemos começar com um chute inicial que chamamos de "chute malvado ou chute longe" pois para que o algoritmo consiga alcançar o x^* teria que passar por uma região adversa (um vale, por exemplo), com isso poderíamos testar a robustez do método.

- As dificuldades de resolução dos problemas na prática podem existir pois os cálculos dos gradientes, normas, valor da imagem $f(x)$, distância entre pontos, etc, dependem muito da precisão da máquina ou compilador testados, que por sua vez é finita. Este fato ocorre, por exemplo, quando trabalhamos com uma função mal escalonada (Powell Badly Scaled) e que tomando o ponto $(1.09815933^{-5}; 9.106146738)$ me daria uma $f(x)$ pouco maior que 10^{-8}, ou seja, um ponto pertíssimo de x^* mas, que ainda assim não satisfaria o *critério de parada* sendo satisfeito com estimativa de $f(x) \approx 10^{-13}$. Isto acontece também com algumas funções que falaremos mais a frente.

- A convergência de um problema pode ser constatada quando a norma-2 do gradiente no ponto examinado for menor que um valor suficientemente pequeno (que assumimos como sendo $\epsilon = 10^{-5}$). Porém, quando sua convergência não for possível (em muitos casos devido a precisão da máquina ou do compilador) adotamos duas formas de abordar tal situação. A primeira delas o NC (não converge) dizemos que isto acontece quando o problema ultrapassou as $k = 50.000$ iterações (o valor da norma-2 não alcançou o ϵ desejado). A segunda situação é o D que acontece quando o problema diverge (a norma-2 tende a $+\infty$ pois a precisão do compilador não suportaria tal

valor).

Os problemas serão definidos e analisados em seções a seguir.

4.2 Funções Quadráticas

Os primeiros problemas testes implementados foram de funções quadráticas, as quais não pertencem ao conjunto de problemas propostos por [3], mas constituem um teste de validação da implementação. Os resultados que se obtêm com o método dos Gradientes Conjugados seguem conforme descrevemos no capítulo anterior. Para qualquer chute inicial que introduzimos como entrada, a convergência se dá em no máximo n iterações quando trabalhamos com busca linear exata, mas com outra busca linear, por exemplo, Armijo este fato não necessariamente é confirmado pois dependendo da complexidade (condicionamento da matriz A) de A o número de iterações pode aumentar, isto é, é diretamente relacionado com o condicionamento dessa matriz e este comportamento é possível perceber nas Tabelas 4.1 e 4.4.

Problema 1: Quadrática

1a: Hessiana=Identidade

Este problema é o mais clássico dos problemas quadráticos. Ele apresenta uma Hessiana particular cuja forma é exatamente a da matriz Identidade que por esta razão tem a característica de ter convergência em máximos n passos qualquer que seja a busca linear utilizada, os resultados são exibidos na Tabela 4.1.

Dimensão: $m = n$
Funções: $f_i(x) = x_i$
Ponto inicial: $x^0 = (x_j^0)$ onde $x_{2j-1}^0 = 10$; $x_{2j}^0 = 3 \ \ \forall j = 1, \cdots, n/2$
Minimizador: $f = 0$ com $(0; \ldots; 0)$

Quadrática						
Dimensão	2		12		36	
Beta	FR	PR	FR	PR	FR	PR
Ponto que converge	$x_1 = 0$; $x_2 = 0$		$x_j = 0 \, \forall j$		$x_j = 0 \, \forall j$	
Número iterações	2	2	2	2	2	2
Norma	0	0	0	0	0	0

Tabela 4.1: Quadrática dim=(2/12/36) ; $x_j^0 = (x_{2j-1}^0 = 10 \ ; \ x_{2j}^0 = 3)$

1b: Quadrática 2

Seja a função quadrática da forma,

$$\begin{aligned} f(x) &= \frac{1}{2} x^T A x + b^T x. \\ \nabla f(x) &= Ax + b. \end{aligned}$$

Podemos variar o problema quadrático apenas alterando a matriz Hessiana A, e fazemos da seguinte maneira:

- I) $A = H := diag(m, m-1, \cdots, 1)$; $b = (1, \cdots, 1)^T$; $x^* = (0, \cdots, 0)^T$;
- II) $A = H := diag(10m, 5m, m, m-1, \cdots, 1)$; $b = (1, \cdots, 1)^T$; $x^* = (0, \cdots; 0)^T$;

com $m = 10$, 100 e 1000.

Quadrática 2 - Problema I						
Dimensão	10		100		1000	
Beta	FR	PR	FR	PR	FR	PR
Ponto que converge	$x_j = 0 \forall j$		$x_j = 0 \forall j$		$x_j = 0 \forall j$	
Número iterações	3	3	3	3	3	3
Norma	0	0	0	0	0	0

Tabela 4.2: Quadrática 2; Problema I; **Busca Linear: Exata**

Quadrática 2 - Problema I						
Dimensão	10		100		1000	
Beta	FR	PR	FR	PR	FR	PR
x^0	$x_j^0 = 1$					
Número iterações	D	44.325	D	42.661	D	D
Ponto que converge	$x_j = 0 \forall j$		$x_j = 0 \forall j$		$x_j = 0 \forall j$	
x^0	$x_j^0 = 0.2$					
Número iterações	D	44.051	D	44.485	D	D
Ponto que converge	$x_j = 0 \forall j$		$x_j = 0 \forall j$		$x_j = 0 \forall j$	

Tabela 4.3: Quadrática 2; Problema I; **Busca Linear: Armijo**

Quadrática 2 - Problema II						
Dimensão	10		100		1000	
Beta	FR	PR	FR	PR	FR	PR
Ponto que converge	$x_j = 0 \, \forall j$		$x_j = 0 \, \forall j$		$x_j = 0 \, \forall j$	
Número iterações	3	3	3	3	3	3
Norma	0	0	0	0	0	0

Tabela 4.4: Quadrática 2; Problema II; **Busca Linear: Exata**

Quadrática 2 - Problema II						
Dimensão	10		100		1000	
Beta	FR	PR	FR	PR	FR	PR
x^0	$x_j^0 = 1$					
Número iterações	D	40.532	D	44.438	D	D
Ponto que converge	$x_j = 0 \, \forall j$		$x_j = 0 \, \forall j$		$x_j = 0 \, \forall j$	
x^0	$x_j^0 = 0.2$					
Número iterações	726	42.675	D	43.559	D	D
Ponto que converge	$x_j = 0 \, \forall j$		$x_j = 0 \, \forall j$		$x_j = 0 \, \forall j$	

Tabela 4.5: Quadrática 2; Problema II; **Busca Linear: Armijo**

Os resultados expostos nas tabelas anteriores exibem características já mencionadas em capítulos anteriores. O número condicional da matriz Hessiana está diretamente ligado a complexidade que o algoritmo tem de resolver o problema. Podemos perceber que conforme aumentamos a dimensão de A, maior é a taxa de convergência.

4.3 Função Rosenbrock

O segundo problema teste consiste na minimização da função de Rosenbrock. Esta função é uma função muito conhecida que tem uma "garganta" muito comprida com uma "parede" muito íngrime e um fundo quase plano. Ela é uma função bem complexa pois é extremamente sensível ao chute inicial que quando adotado muito longe do minimizador x^* pode causar a divergência do método, também podemos citar que por menor que seja a distância dos pontos, produz uma diferença considerável do valor da imagem $f(x)$.

Por exemplo, se escolhermos o chute $x_1 = 10^8$ e $x_2 = x_1^2$, os dois gradientes sucessivos são exatamente o oposto um do outro (até mesmo com tamanho de passo α bem pequeno), isto significa que não poderia alcançar o "fundo" por causa da finita precisão de cálculos do computador e assim não poderia calcular uma direção d_k certa para ir ao longo da "garganta". Por este motivo, qualquer método que utilize o gradiente da função falharia para minimizar esta função quando adotados chute inicial longe do x^*.

A função de Rosenbrock é também chamada de função Banana, por causa do gráfico das curvas de nível. Ela apresenta um único minimizador (local=global) e Moré [3] adota o chute inicial $x^0 = (-1.2; 1)$ para que o algoritmo tente buscar a solução passando por um grande vale até alcançar o minimizador x^*.

Com este problema vamos buscar a solução com as dimensões 2, 12 e 36, conforme foi mencionado.

A primeira análise que podemos fazer neste problema é que, por exemplo, para a dimensão 2 se entrarmos com um chute inicial afastado da solução, o β^{PR} converge para a solução do problema, ou seja, se comporta muito bem, enquanto que o β^{FR} sequer converge (ver Tabela 4.6). Entretanto, quando tratamos o problema na mesma dimensão com um chute inicial relativamente próximo da solução, o β^{FR} converge e converge expressivamente mais rápido do que o β^{PR} (comparando o número de iterações - ver Tabela 4.7).

Problema 2: Rosenbrock

Dimensão:	$m = n$
Funções:	$f_{2i-1}(x) = 10(x_{2i} - x_{2i-1}^2)$
	$f_{2i}(x) = 1 - x_{2i-1}\ \forall i = 1, \cdots, m/2$
Ponto inicial:	$x^0 = (x_j^0)$ onde $x_{2j-1}^0 = -1,2\ ;\ x_{2j}^0 = 1\ \forall j = 1, \cdots, n/2$
Minimizador:	$f = 0$ com $(1; \ldots; 1)$

Para esta função, a Jacobiana é uma matriz diagonal em bloco da seguinte forma:

$$J(x) = \begin{pmatrix} J_1(x) & 0 & \cdots & 0 \\ 0 & J_2(x) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & J_{n/2}(x) \end{pmatrix},$$

$$J_i(x) = \begin{pmatrix} -20x_{2i-1} & -1 \\ 10 & 0 \end{pmatrix}.$$

A seguir os resultados obtidos com o Método dos Gradientes Conjugados implementado.

Rosenbrock		
Dimensão	2	
Beta	FR	PR
Ponto que converge	D	$x_1 = 0,999989$ $x_2 = 0,999978$
Valor imagem	D	$f(x) = 1,252 * 10^{-10}$
Número iterações	D	215.554*
Norma	$+\infty$	10^{-5}
Tempo execução (seg)	D	193

Tabela 4.6: Rosenbrock dim=2 ; $x^0 = (-1,2 \quad ; \quad 1,0)$

$*$ O número máximo de iteração (50.000) foi alcançado para o teste com β^{PR} mas, percebemos que o ponto atingido estava muito próximo de x^* (1,1) então permitimos que o algoritmo ultrapassasse a iteração máxima até que o algoritmo atingisse o critério de parada ($||g|| = 10^{-5}$) nos resultando 215.554 iterações para alcançar o minimizador.

As características encontradas em Rosenbrock com dimensão 2 se estendem também ao problema de Rosenbrock *Extended*, ou seja, ao Rosenbrock com dimensões maiores que 2. O Rosenbrock *Extended* se comporta muito semelhante ao problema de dimensão

Rosenbrock		
Dimensão	2	
Beta	**FR**	**PR**
Ponto que converge	$x_1 = 0,999989$ $x_2 = 0,999978$	$x_1 = 0,986600$ $x_2 = 0,999978$
Valor imagem	$f(x) = 1,21 * 10^{-10}$	$f(x) = 1,252 * 10^{-10}$
Número iterações	1.503	194.980*
Norma	10^{-5}	10^{-5}
Tempo execução (seg)	2	166

Tabela 4.7: Rosenbrock dim=2 ; $x^0 = (0,7 \quad ; \quad 0,7)$

2 quando entramos com um chute inicial afastado da solução. Mas, quando utilizamos um chute inicial próximo da solução, o comportamento do desempenho com os β não se diferenciam muito, e o β^{PR} tem um comportamento ligeiramente melhor, conforme pode ser verificado nas Tabelas 4.8 e 4.9 seguintes:

Rosenbrock-Extended				
Dimensão	12		36	
Beta	**FR**	**PR**	**FR**	**PR**
Ponto que converge	D	$x = (x_j^0)$ $x_{2j-1}^0 = 0,999997$ $x_{2j}^0 = 0,999995$	D	$x = (x_j^0)$ $x_{2j-1}^0 = 0,999997$ $x_{2j}^0 = 0,999995$
Valor imagem	D	$f(x) = 1,09 * 10^{-10}$	D	$f(x) = 1,09 * 10^{-10}$
Número iterações	D	19.231	D	19.252
Norma	$+\infty$	10^{-5}	$+\infty$	10^{-5}
Tempo execução (seg)	D	113	D	261

Tabela 4.8: Rosenbrock dim=(12 / 36) ; $x^0 = (x_{2j-1}^0 = -1,2 \quad ; \quad x_{2j}^0 = 1)$

Rosenbrock-Extended				
Dimensão	12		36	
Beta	FR	PR	FR	PR
Ponto que converge	$x = (x_j^0)$ $x_{2j-1}^0 = 0,999995$ $x_{2j}^0 = 0,999991$	$x = (x_j^0)$ $x_{2j-1}^0 = 0,999996$ $x_{2j}^0 = 0,999991$	$x = (x_j^0)$ $x_{2j-1}^0 = 0,999997$ $x_{2j}^0 = 0,999995$	$x = (x_j^0)$ $x_{2j-1}^0 = 0,999998$ $x_{2j}^0 = 0,999995$
Valor imagem	$f(x) = 1,16 * 10^{-10}$	$f(x) = 1,16 * 10^{-10}$	$f(x) = 1,09 * 10^{-10}$	$f(x) = 4 * 10^{-12}$
Número iterações	1.644	1.156	1.723	1.215
Norma	10^{-5}	10^{-5}	10^{-5}	10^{-5}
Tempo execução (seg)	9	6	23	16

Tabela 4.9: Rosenbrock dim=(12 / 36) ; $x^0 = (0,7; \ldots; 0,7)$

4.4 Função Freudenstein-Roth

A função de Freudenstein e Roth tem como minimizadores dois pontos distintos. Enquanto que $(11,412787\,;\, -0,896805)$ nos dá uma imagem $f(x) = 48,98425367$, o ponto $(5,0\,;\, 4,0)$ nos retorna $f(x) = 0$, ou seja, a função Freudenstein e Roth possui um mínimo local e outro global. O problema Freudenstein-Roth é um problema no qual o algoritmo implementado apresentou um comportamento peculiar. Nele podemos constatar que a escolha do β pode ter grande influência no desempenho do algoritmo, reforçando o que foi mencionado no capítulo anterior que para problemas não-quadráticos a escolha deste parâmetro pode modificar o desempenho do método.

Nas Tabelas 4.10 e 4.11 vemos que quando entramos com um chute inicial afastado da solução o β^{FR} converge para um mínimo local e em contrapartida o β^{PR} converge para o mínimo global. Esse comportamento pode ser melhor compreendido por meio da Figura 4.2, em que observamos o desenho da superfície da função e o caminho descrito pelo método com a utilização dos dois β's.

Ao entrarmos com um chute inicial mais próximo da solução, o β^{FR} converge também para o mínimo global assim como o β^{PR}, porém com uma convergência bem mais rápida.

A Figura 4.1 representa o gráfico da função Freudenstein-Roth para o domínio [-15;40]x[-

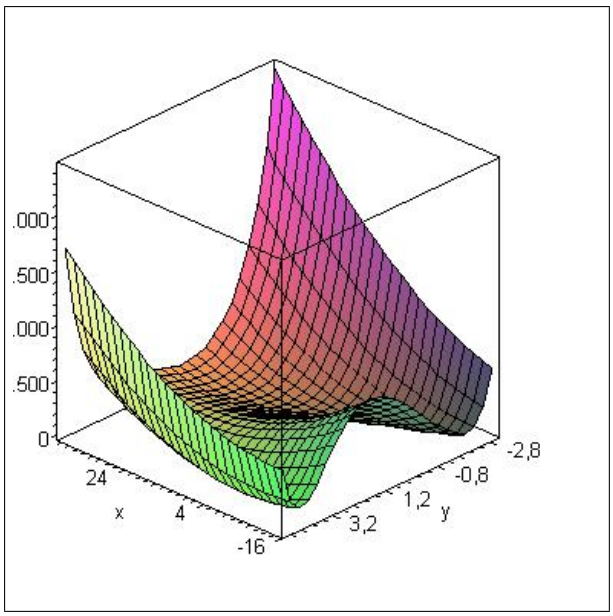

Figura 4.1: *Gráfico da função Freudenstein-Roth*

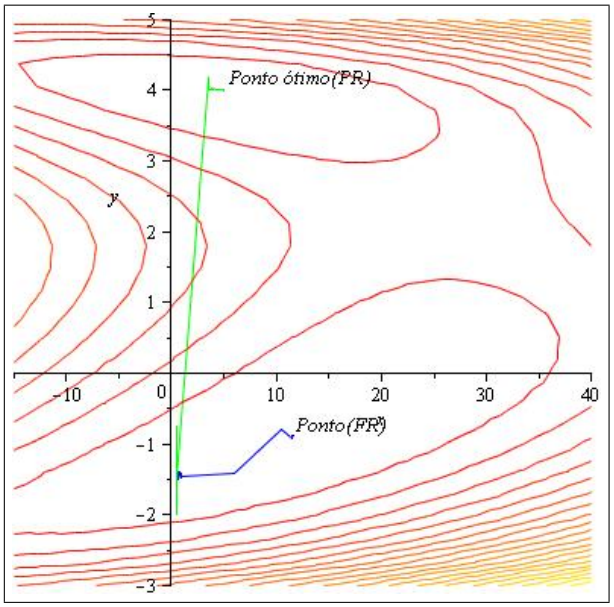

Figura 4.2: *Curvas de nível função Freudenstein-Roth*

3;5]. Na Figura 4.2 é ilustrado as curvas de nível desta função juntamente com o desenvolvimento dos caminhos percorridos que os iterandos fazem até encontrar a solução, sendo que o percurso de cor verde é o caminho percorrido com a utilização do método com o β^{PR} e o de cor azul, o caminho percorrido com a utilização do β^{FR}.

Problema 3: Freudenstein-Roth

Dimensão: $n = 2$ e $m = 2$
Funções: $f_1(x) = -13 + x_1 + ((5 - x_2)x_2 - 2)x_2$
$f_2(x) = -29 + x_1 + ((x_2 + 1)x_2 - 14)x_2$
Ponto inicial: $x^0 = (0,5\ ;\ -2)$
Minimizador: $f = 0$ com $(5; 4)$

Freudenstein-Roth		
Dimensão	2	
Beta	**FR**	**PR**
Ponto que converge	$x_1 = 11,412787$ $x_2 = -0,896805$	$x_1 = 5,000002$ $x_2 = 4,000000$
Valor imagem	$f(x) = 48,98425367$	$f(x) = 8 * 10^{-12}$
Número iterações	69	371
Norma	10^{-5}	10^{-5}
Tempo execução (seg)	1	2

Tabela 4.10: Freudenstein-Roth dim=2 ; $x^0 = (0,5\ \ ;\ \ -2)$

Freudenstein-Roth		
Dimensão	2	
Beta	**FR**	**PR**
Ponto que converge	$x_1 = 5,000002$ $x_2 = 4,000000$	$x_1 = 4,999999$ $x_2 = 4,000000$
Valor imagem	$f(x) = 8 * 10^{-12}$	$f(x) = 2 * 10^{-12}$
Número iterações	54	459
Norma	$5 * 10^{-6}$	$9 * 10^{-6}$
Tempo execução (seg)	1	2

Tabela 4.11: Freudenstein-Roth dim=2 ; $x^0 = (4,5\ \ ;\ \ 3,5)$

4.5 Função Powell

A função seguinte que descreveremos tem seu desempenho muito parecido com o problema de Rosenbrock-Extended quando se trata da questão do chute inicial, tanto com a menor dimensão suportada pelo problema (a saber, dimensão 4), quanto nas outras dimensões. Todavia, neste problema algumas peculiaridades aparecem, pois quando partimos de um ponto afastado da solução o comportamento do β^{PR} não tende a convergência num número confortável de iterações, já o β^{FR} mantém uma atuação esperada.

Os resultados obtidos quando entramos com chute inicial próximo da solução tende a se comportar de forma já conhecida porém, há um pequeno destaque do β^{FR} sobre o β^{PR} na velocidade de convergência.

Problema 4: Powell-Extended-Singular

Dimensão:	n =múltiplo de 4 e $m = n$
Funções:	$f_{4i-3}(x) = x_{4i-3} + 10x_{4i-2} \;\; \forall\, i, j$
	$f_{4i-2}(x) = \sqrt{5}(x_{4i-1} - x_{4i})$
	$f_{4i-1}(x) = (x_{4i-2} - 2x_{4i-1})^2$
	$f_{4i}(x) = \sqrt{10}(x_{4i-3} - x_{4i})^2$
Ponto inicial:	$x^0 = (x_j^0)$ onde $\quad x_{4j-3}^0 = 3\ ;\ x_{4j-2}^0 = -1\ ;\ x_{4j-1}^0 = 0\ ;\ x_{4j}^0 = 1$
Minimizador:	$f = 0$ com $\quad (0; \ldots; 0)$

A matriz Jacobiana J(x) para $n = 4$ é:

$$J(x) = \begin{pmatrix} 1 & 10 & 0 & 0 \\ 0 & 0 & \sqrt{5} & -\sqrt{5} \\ 0 & 2(x_2 - 2x_3) & -4(x_2 - 2x_3) & 0 \\ 2\sqrt{10}(x_1 - x_4) & 0 & 0 & -2\sqrt{10}(x_1 - x_4) \end{pmatrix}.$$

Powell-Extended-Singular		
Dimensão	4	
Beta	**FR**	**PR**
Ponto que converge	D	$x_1 = 0.225206$; $x_2 = -0.022375$ $x_3 = 0.086194$; $x_4 = 0.094828$
Valor imagem	D	$f(x) = 4,703197648 * 10^{-3}$
Número iterações	D	50.000
Norma	$+\infty$	NC(0.0275)
Tempo execução (seg)	D	115

Tabela 4.12: Powell-Extended-Singular dim=4 ; $x^0 = (3; -1; 0; 1)$

Powell-Extended-Singular		
Dimensão	4	
Beta	**FR**	**PR**
Ponto que converge	$x_1 = 0.009901$; $x_2 = -0.000990$ $x_3 = 0.004999$; $x_4 = 0.004999$	$x_1 = 0.009901$; $x_2 = -0.000990$ $x_3 = 0.004998$; $x_4 = 0.004998$
Valor imagem	$f(x) = 2,035243515 * 10^{-8}$	$f(x) = 2,034653803 * 10^{-8}$
Número iterações	203	3.960
Norma	10^{-5}	10^{-5}
Tempo execução (seg)	1	4

Tabela 4.13: Powell-Extended-Singular dim=4 ; $x^0 = (0,01\ ; 0\ ; 0,01\ ; 0)$

Powell-Extended-Singular		
Dimensão	12	
Beta	FR	PR
Ponto que converge	D	$x = (x_j)$ $x_{4j-3} = 0.225206$; $x_{4j-2} = -0.022375$ $x_{4j-1} = 0.086194$; $x_{4j} = 0.094828$
Valor imagem	D	$f(x) = 4,703197648 * 10^{-3}$
Número iterações	D	50.000
Norma	$+\infty$	NC(0.047632)
Tempo execução (seg)	D	160

Tabela 4.14: Powell Ext.Sing. dim=12 ; $x_j^0 = (x_{4j-3}^0 = 3\,; x_{4j-2}^0 = -1\,; x_{4j-1}^0 = 0\,; x_{4j}^0 = 1)$

Powell-Extended-Singular		
Dimensão	12	
Beta	FR	PR
Ponto que converge	$x = (x_j)$ $x_{4j-3} = 0.009901$; $x_{4j-2} = -0.000990$ $x_{4j-1} = 0.004998$; $x_{4j} = 0.004999$	$x = (x_j)$ $x_{4j-3} = 0.009901$; $x_{4j-2} = -0.000990$ $x_{4j-1} = 0.004998$; $x_{4j} = 0.004998$
Valor imagem	$f(x) = 2,035243515 * 10^{-6}$	$f(x) = 2,035243515 * 10^{-6}$
Número iterações	221	4.300
Norma	10^{-5}	10^{-5}
Tempo execução (seg)	11	49

Tabela 4.15: Powell Ext.Sing. dim=12 ; $x_j^0 = (x_{4j-3}^0 = 0,01\,; x_{4j-2}^0 = 0\,; x_{4j-1}^0 = 0,01\,; x_{4j}^0 = 0)$

Powell-Extended-Singular		
Dimensão	36	
Beta	FR	PR
Ponto que converge	D	$x = (x_j)$ $x_{4j-3} = 0.225206$; $x_{4j-2} = -0.022375$ $x_{4j-1} = 0.086194$; $x_{4j} = 0.094828$
Valor imagem	D	$f(x) = 4,703197648 * 10^{-3}$
Número iterações	D	50.000
Norma	$+\infty$	NC(0.0825)
Tempo execução (seg)	D	157

Tabela 4.16: Powell Ext.Sing. dim=36 ; $x_j^0 = (x_{4j-3}^0 = 3\,; x_{4j-2}^0 = -1\,; x_{4j-1}^0 = 0\,; x_{4j}^0 = 1)$

Powell-Extended-Singular		
Dimensão	36	
Beta	FR	PR
Ponto que converge	$x = (x_j)$ $x_{4j-3} = 0.009869$; $x_{4j-2} = -0.000987$ $x_{4j-1} = 0.004827$; $x_{4j} = 0.004828$	$x = (x_j)$ $x_{4j-3} = 0.009878$; $x_{4j-2} = -0.000988$ $x_{4j-1} = 0.004880$; $x_{4j} = 0.004880$
Valor imagem	$f(x) = 1,928476772 * 10^{-6}$	$f(x) = 1,953597131 * 10^{-6}$
Número iterações	1.303	50.000
Norma	10^{-5}	0.000025
Tempo execução (seg)	15	226

Tabela 4.17: Powell Ext.Sing. dim=36 ; $x_j^0 = (x_{4j-3}^0 = 0,01\,; x_{4j-2}^0 = 0\,; x_{4j-1}^0 = 0,01\,; x_{4j}^0 = 0)$

4.6 Função Broyden Tridiagonal

Esta função possui alguns mínimos locais para qualquer dimensão aceitável por ela. A função de Broyden Tridiagonal é um problema que traz uma característica diferenciada dos outros problemas em relação ao desempenho dos beta's sobre a questão do ponto inicial atribuído por Moré [3].

Podemos notar que para todas as funções aqui testadas deste conjunto, somente uma dessas funções, além da função de Broyden, conseguiu uma boa execução para os dois beta's com o chute inicial dado por Moré [3], sendo que quando nos afastamos da solução tal problema se comporta de forma esperada - o algoritmo com o β^{PR} converge.

Problema 5: Broyden Tridiagonal

Dimensão:	$n \geq 3$ e $m = n$
Funções:	$f_i(x) = (3 - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1$
	onde $x_0 = x_{n+1} = 0$
Ponto inicial:	$x^0 = (-1; \ldots; -1)$
Minimizador:	$f = 0$

Sua Jacobiana J(x) é uma matriz tridiagonal expressa da seguinte maneira:

$$J(x) = \begin{pmatrix} 3-4x_1 & -2 & 0 & 0 & \cdots & 0 \\ -1 & 3-4x_2 & -2 & 0 & \cdots & 0 \\ 0 & -1 & 3-4x_3 & -2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -1 & 3-4x_{n-1} & -2 \\ 0 & 0 & \cdots & 0 & -1 & 3-4x_n \end{pmatrix}.$$

Broyden Tridiagonal		
Dimensão	3	
Beta	**FR**	**PR**
Ponto que converge	$x_1 = -0.526773$; $x_2 = -0.567649$ $x_3 = -0.410313$	$x_1 = -0.526773$; $x_2 = -0.567649$ $x_3 = -0.410313$
Valor imagem	$f(x) = 2,7 * 10^{-12}$	$f(x) = 2,9 * 10^{-12}$
Número iterações	304	6.327
Norma	$9 * 10^{-6}$	10^{-5}
Tempo execução (seg)	1	6

Tabela 4.18: Broyden Tridiagonal dim=3 ; $x^0 = (-1\,;-1\,;-1)$

Broyden Tridiagonal		
Dimensão	3	
Beta	**FR**	**PR**
Ponto que converge	D	$x_1 = -0.526773$; $x_2 = -0.567649$ $x_3 = -0.410312$
Valor imagem	D	$f(x) = 1,4209901 * 10^{-11}$
Número iterações	D	9.592
Norma	$+\infty$	10^{-5}
Tempo execução (seg)	D	5

Tabela 4.19: Broyden Tridiagonal dim=3 ; $x^0 = (-2\,;1\,;-0,5)$

Broyden Tridiagonal		
Dimensão	12	
Beta	**FR**	**PR**
Ponto que converge	**x**	**x**
Valor imagem	$f(x) = 2,8 * 10^{-12}$	$f(x) = 2,8 * 10^{-12}$
Número iterações	341	7.250
Norma	10^{-5}	10^{-5}
Tempo execução (seg)	2	10

Tabela 4.20: Broyden Tridiagonal dim=12 ; $x^0 = (-1; \ldots; -1)$

Ponto de convergência

x $\Rightarrow$ $x_1 = -0.570756; x_2 = -0.681896; x_3 = -0.702449; x_4 = -0.706160; x_5 = -0.706678; x_6 = -0.706329; x_7 = -0.705057; x_8 = -0.701524; x_9 = -0.691895; x_{10} = -0.665798; x_{11} = -0.596036; x_{12} = -0.416412$.

Broyden Tridiagonal		
Dimensão	12	
Beta	**FR**	**PR**
Ponto que converge	D	**x**
Valor imagem	D	$f(x) = 1,0771010275968$
Número iterações	D	50.000
Norma	∞	NC(0.024451)
Tempo execução (seg)	D	117

Tabela 4.21: Broyden Tridiagonal dim=12 ; $x_j^0 = (x_{3j-2}^0 = -2\ ;\ x_{3j-1}^0 = 1\ ;\ x_{3j}^0 = -0,5)$

Ponto de convergência

x $\Rightarrow$ $x_1 = 0.217497; x_2 = 0.789268; x_3 = 0.973806; x_4 = 0.592582; x_5 = 0.529292; x_6 = 0.757106; x_7 = 0.876096; x_8 = 0.584255; x_9 = 0.484372; x_{10} = 0.794128; x_{11} = 1.141703; x_{12} = 0.263966$.

Broyden Tridiagonal		
Dimensão	36	
Beta	FR	PR
Ponto que converge	D	x
Valor imagem	D	$f(x) = 5,79359407$
Número iterações	D	23.146
Norma	∞	10^{-5}
Tempo execução (seg)	D	134

Tabela 4.22: Broyden Tridiagonal dim=36 ; $x^0 = (-1; \ldots; -1)$

Ponto de convergência

x $\Rightarrow$ $x[1] = 0.652401; x[2] = -0.619455; x[3] = -0.748711; x[4] = -0.735378; x[5] = -0.719831; x[6] = -0.712204; x[7] = -0.709054; x[8] = -0.707835; x[9] = -0.707376; x[10] = -0.707206; x[11] = -0.707143; x[12] = -0.707120; x[13] = -0.707112; x[14] = -0.707109; x[15] = -0.707107; x[16] = -0.707107; x[17] = -0.707107; x[18] = -0.707107; x[19] = -0.707107; x[20] = -0.707107; x[21] = -0.707106; x[22] = -0.707106; x[23] = -0.707104; x[24] = -0.707099; x[25] = -0.707085; x[26] = -0.707046; x[27] = -0.706941; x[28] = -0.706653; x[29] = -0.705866; x[30] = -0.703712; x[31] = -0.697806; x[32] = -0.681570; x[33] = -0.636821; x[34] = -0.513728; x[35] = -0.183705; x[36] = 0.598780.$

Broyden Tridiagonal		
Dimensão	36	
Beta	FR	PR
Ponto que converge	D	x
Valor imagem	D	$f(x) = 6,08350828$
Número iterações	D	50.000
Norma	∞	NC(0.175107)
Tempo execução (seg)	D	381

Tabela 4.23: Broyden Tridiagonal dim=36 ; $x_j^0 = (x_{3j-2}^0 = -2\ ;\ x_{3j-1}^0 = 1\ ;\ x_{3j}^0 = -0,5)$

Ponto de convergência

x $\Rightarrow$ $x[1] = 0.650482; x[2] = -0.599390; x[3] = -0.685594; x[4] = -0.549725; x[5] = -0.221928; x[6] =$

$0.424187; x[7] = 1.082397; x[8] = 0.726062; x[9] = 0.519099; x[10] = 0.652166; x[11] = 0.796360; x[12] = 0.731439; x[13] = 0.662770; x[14] = 0.690307; x[15] = 0.728268; x[16] = 0.716646; x[17] = 0.696951; x[18] = 0.700865; x[19] = 0.711547; x[20] = 0.711989; x[21] = 0.705906; x[22] = 0.701653; x[23] = 0.705310; x[24] = 0.715225; x[25] = 0.713494; x[26] = 0.693084; x[27] = 0.691236; x[28] = 0.730904; x[29] = 0.743413; x[30] = 0.662250; x[31] = 0.628378; x[32] = 0.766893; x[33] = 0.881934; x[34] = 0.551978; x[35] = 0.372380; x[36] = 0.696957$.

4.7 Função Himmelblau

A função de Himmelblau é uma função cujas soluções são os pontos $(3,584428\,;\,-1,848126)$, $(3,0\,;\,2,0)$, $(-2,805118\,;\,3,131313)$ e $(-3,779310\,;\,-3,283186)$ com valores de imagens $f(x) = 0.8883044e-11$, $f(x) = 0$, $f(x) = 0.9634916e-11$ e $f(x) = 0.37825e-11$, respectivamente. Assim como a função Freudenstein-Roth, esta última função que apresentamos também nos traz um importante resultado enfatizando os diferentes desempenhos do método quando alteramos o parâmetro que regula a direção d no método, ou seja, a variável β. No domínio analisado existem quatro minimizadores e um deles é o minimizador global. Quando fazemos uso do β^{FR} o minimizador encontrado pelo método resulta num minimizador local, enquanto que ao utilizarmos o β^{PR} resulta no minimizador global. Veremos tal processo na figura seguinte.

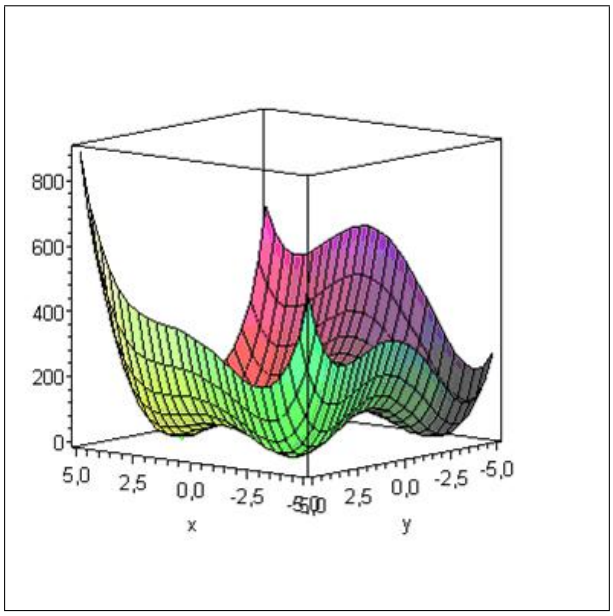

Figura 4.3: *Gráfico da função Himmelblau*

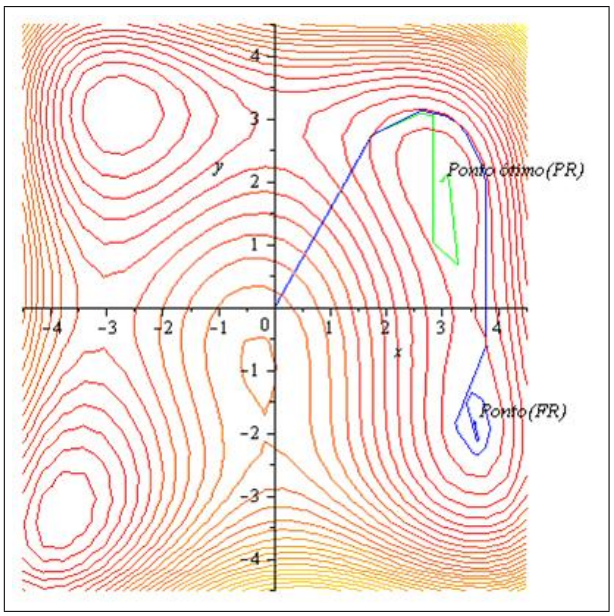

Figura 4.4: *Curvas de nível função Himmelblau*

Problema 6

Dimensão:	$n = 2$ e $m = 2$
Funções:	$f_1(x) = x_1^2 + x_2 - 11$
	$f_2(x) = x_1 + x_2^2 - 7$
Ponto inicial:	$x^0 = (0\ ;\ 0)$
Minimizador:	$f = 0$ com $(3; 2)$

Função Himmelblau		
Dimensão	2	
Beta	**FR**	**PR**
Ponto que converge	$x_1 = 3,584428$ $x_2 = -1,848126$	$x_1 = 3,00000$ $x_2 = 2,000000$
Valor imagem	$f(x) = 1,6 * 10^{-12}$	$f(x) = 2 * 10^{-13}$
Número iterações	63	31
Norma	10^{-5}	$4 * 10^{-6}$
Tempo execução (seg)	1	1

Tabela 4.24: Função Himmelblau dim=2 ; $x^0 = (0,0 \quad ; \quad 0,0)$

Função Himmelblau		
Dimensão	2	
Beta	**FR**	**PR**
Ponto que converge	$x_1 = 3,000000$ $x_2 = 2,000000$	$x_1 = 3,000000$ $x_2 = 2,000000$
Valor imagem	$f(x) = 10^{-12}$	$f(x) = 6 * 10^{-13}$
Número iterações	28	27
Norma	$9 * 10^{-6}$	10^{-5}
Tempo execução (seg)	1	1

Tabela 4.25: Função Himmelblau dim=2 ; $x^0 = (2,5 \quad ; \quad 1,5)$

4.8 Discussões

Dos resultados mostrados neste capítulo percebemos portanto, que o β^{PR} é melhor quando o chute inicial está longe do minimizador e o β^{FR} tem melhor desempenho quando o chute inicial está próximo do minimizador x^*.

4.9 Validação do Código

Há a necessidade de constatar se o código implementado realmente condiz com a realidade dos diversos estudos que temos sobre o Método dos Gradientes Conjugados. E por isto, buscamos por materiais de estudiosos que trabalharam com este método, cujos resultados são expostos através de dados como tabelas, gráficos, etc, fazendo assim uma comparação entre estes resultados e os resultados aqui obtidos para assim validarmos o nosso código.

Uma validação "extrai" todas as variáveis em que o autor Mikael [22] (que escolhemos como referência) insere ao seu código tais como problema analisado, critério de parada, busca linear, β, precisão da máquina testada, etc e replica essas variáveis no código que implementamos fazendo assim uma comparação desses resultados.

Validação Código				
Problema Quadrático 2				
Busca Linear Exata; $\epsilon = 10^{-5}$; $x^0 = (1, ..., 1)$				
Referências	Mikael [22]		Em Anexo	
Beta	**FR**	**PR**	**FR**	**PR**
Número iterações	145	145	3	3

Tabela 4.26: $x^0 = (1,0 \quad ; \quad 1,0)$

Validação Código		
Rosenbrock - Dimensão 2		
$\epsilon = 10^{-6}$; $x^0 = (0,99; 0,99)$		
Código testado	Mikael [22]	Em Anexo
Beta	**FR**	**FR**
Número iterações	478	1.022

Tabela 4.27: $x^0 = (0,0 \quad ; \quad 0,0)$

As Tabelas 4.26 e 4.27 expõe o número de iterações resultantes dos testes feitos com as funções Quadrática 2 e a função Rosenbrock (Dimensão 2) para o nosso código e o número de iterações obtidos por Mikael Fallgren [22]. Ou seja, se referem aos resultados obtidos da implementação do Algoritmo 5 com essas duas funções, replicando as diretrizes (mesmos chute inicial, critério de parada, busca linear, etc) em nosso código para comparar com os valores obtidos por Mikael [22].

Apesar dos valores encontrados por [22] (em Tabela 4.27) estarem um pouco distante dos valores encontrados pelo nosso código, ainda assim podemos dizer que este código é satisfatório visto que é extremamente árduo encontrarmos um pesquisador que já tenha trabalhado com o método dos Gradientes Conjugado, com este mesmo problema (Rosenbrock) e com todas as variáveis que influenciam o código como: chute inicial, busca linear (exata, backtracking, Armijo, Wolfe), precisão adotada (critério de parada - ϵ) e as constantes c_1 e c_2 (caso este pesquisador tenha usado busca linear com as condições forte de Wolfe) além de outras.

Capítulo 5

O Híbrido

O capítulo anterior trouxe através de tabelas e gráficos como o método dos Gradientes Conjugados se comporta com algumas classes de funções. Nele fazemos uso de ferramentas matemáticas como: gradiente da função, hessiana da função, produto interno, etc, para nos auxiliar a encontrarmos padrões do o por quê o método comportar-se diferentemente quando variamos o β.

Na função Freudenstein-Roth, por exemplo, é possível visualizar claramente que quando assumimos β's diferentes o desenvolvimento do algoritmo varia. Há dois minimizadores nesta função, sendo um minimizador que retorna o valor em que se tem a menor imagem (minimizador global) e outro minimizador não global (minimizador local). Esta função é importante para nossa análise, pois quando adotamos o β^{FR} o método converge para um minimizador enquanto que quando adotamos o β^{PR} o método converge para o outro.

Ao aprofundarmos mais a nossa análise perceberemos alguns dos padrões que fazem com que esta diferença aconteça.

Alguns elementos já estabelecidos pelo método como o vetor gradiente $\nabla f(x_k)$ e os pontos x_k nos mostram alguns fatores importantes que determinam o porquê na diferença de comportamento dos β's e consequentemente nos diferentes resultados encontrados.

Primeiramente vamos tomar o conceito da relação de ângulo entre dois vetores com intuito de esclarecer algumas passagens que virão mais a frente.

$$\cos \ \alpha = \frac{v_1 . v_2}{v_1 v_2} \tag{5.1}$$

Na tabela a seguir é colocado os pontos x_k desde a primeira iteração até a região em que começam-se a diferenciar as soluções.

Soluções x_k Função Himmelblau		
k	Iterandos com β^{FR}	Iterandos com β^{PR}
0	(0,0)	(0,0)
1	(1.75,2.75)	(1.75,2.75)
2	(2.6411114861,3.1369251924)	(2.6050016515,3.0801811667)
3	(2.6411114861,3.1369251924)	(2.8423803125,3.0525053722)
4	(3.0413937118,3.0516854648)	(2.8213681428,1.0347436942)
5	(3.2526477016,2.9348111077)	(3.2656640471,0.6834164070)

Tabela 5.1: Iterandos x_k função Himmelblau

A informação crucial que nos mostra um padrão de comportamento está na disposição dos gradientes da função nos iterados encontrados com o método.

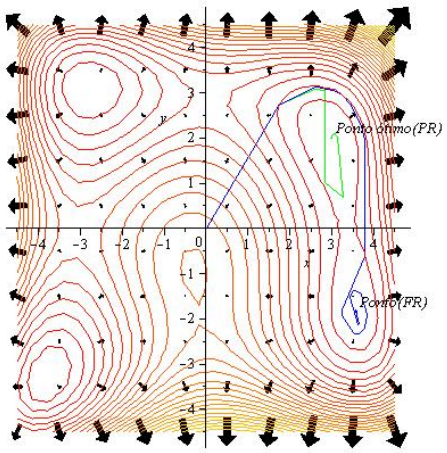

Figura 5.1: Curvas de Nível Função Himmelblau e Gradientes

Vetores gradientes da função Himmelblau		
k	Iterandos com β^{FR}	Iterandos com β^{PR}
1	(-31.687500,15.062500)	(-31.687500,15.062500)
2	(1.585768,67.003897)	(-1.629014,60.475938)
3	(1.585768,67.003897)	(11.816923,63.269041)
4	(26.545029,67.960590)	(-28.844806,-16.873950)
5	(42.447025,62.149449)	(-1.989038,-8.235689)

Tabela 5.2: Gradientes em x_k função Himmelblau

Se focarmos na Figura 5.1 a região onde começa a ocorrer a diferenciação dos β's e descrevermos geometricamente o comportamento dos vetores gradientes $\nabla f(x_k)$ e $\nabla f(x_{k-1)}$, podemos mencionar que quando eles formarem entre si aproximadamente um ângulo de 0° implica que $\nabla f(x_k)$ e $-\nabla f(x_{k-1})$ formarão um ângulo aproximadamente de 180°, implicando que o ângulo θ formado por $\nabla f(x_k)$ e $\nabla f(x_k) - \nabla f(x_{k-1})$ será maior que 90° e assim $cos\,(\theta) < 0$. Com isso,

$$\beta^{PR} = cos\,(\theta)\,\frac{\nabla f(x_k)\ .\ \nabla f(x_k) - \nabla f(x_{k-1})}{\nabla f(x_{k-1})}$$

$$\beta^{PR} < 0.$$

E, sob as mesma condições que as descritas anteriormente o β FR é positivo:

$$\beta^{FR} = \frac{\nabla f(x_k)^{\,2}}{\nabla f(x_{k-1})^{\,2}} > 0.$$

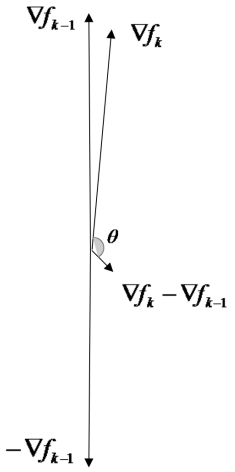

Figura 5.2: Gradientes em x_k e x_{k-1}

Neste capítulo faremos uso desta idéia e proporemos um β que irá conter uma variável ϕ e que ficará atrelada diretamente ao comportamento desse padrão (do cosseno do ângulo formado entre o gradiente atual e o gradiente anterior). E assim demonstraremos mais uma vez que a escolha de diferentes β's interfere no caminho percorrido pelo método, ficando claro uma alteração significativa no desempenho (robustez e eficiência) do método. Este trabalho de dissertação além de apresentar um estudo detalhado do Método dos Gradientes Conjugados, tenta trazer uma pequena contribuição a este fazendo uma fusão dos dois β mais conhecidos da literatura e assim incluir no método um β híbrido com a expectativa de aproveitar as boas características tanto do β^{FR}, quanto do β^{PR}. Ao apresentarmos mais uma vez o β^{FR},

$$= \frac{\nabla f(x_k)^T \nabla f(x_k)}{\nabla f(x_{k-1})^T \nabla f(x_{k-1})} \tag{5.2}$$

e o β^{PR},

$$= \frac{\nabla f(x_k)^T(\nabla f(x_k) - \nabla f(x_{k-1}))}{\nabla f(x_{k-1})^T \nabla f(x_{k-1})} \tag{5.3}$$

percebemos que de (5.3) temos,

$$\begin{aligned} \beta^{PR} &= \frac{\nabla f(x_k)^T \nabla f(x_k)}{\nabla f(x_{k-1})^T \nabla f(x_{k-1})} - \frac{\nabla f(x_k)^T \nabla f(x_{k-1})}{\nabla f(x_{k-1})^T \nabla f(x_{k-1})} \\ \beta^{PR} &= \beta^{FR} - \frac{\nabla f(x_k)^T \nabla f(x_{k-1})}{\nabla f(x_{k-1})^T \nabla f(x_{k-1})} \end{aligned} \tag{5.4}$$

Ao incluirmos uma constante na segunda parcela em (5.4) construímos um novo β.

$$\beta^H = \beta^{FR} - \phi \frac{\nabla f(x_k)^T \nabla f(x_{k-1})}{\nabla f(x_{k-1})^T \nabla f(x_{k-1})} \quad com \quad \phi \in [0,1] \tag{5.5}$$

E assim, quando ϕ for 0, $\beta^H = \beta^{FR}$ e quando ϕ for 1, $\beta^H = \beta^{PR}$.

5.1 O parâmetro ϕ_1

Vimos anteriormente a estratégia que abordaremos de como escolher o valor do parâmetro ϕ. A ideia é fazer com que, se $\nabla f(x_k)$ e $\nabla f(x_{k-1})$ forem aproximadamente múltiplos (paralelos), e vimos no exemplo da seção anterior que os β's terão sinais opostos, alterando a trajetória de convergência do método dos Gradientes Conjugados. A escolha do β nesse caso torna-se muito importante pois é capaz de fazer com que haja convergência para um minimizador global ou local. A partir de testes numéricos realizados com problemas da literatura que possuem seus minimizadores locais e globais conhecidos, observou-se

que na grande maioria dos casos a escolha pelo β^{PR} gerou convergência para o minimizador global (o mesmo fato ocorreu com a função Himmelblau, na Figura 5.1). Sendo assim, os gradientes consecutivos forem aproximadamentes paralelo, implica que ϕ deve receber o valor 1. Analogamente para quando $\nabla f(x_k)$ e $\nabla f(x_{k-1})$ forem aproximadamente ortogonais da fórmula (5.4) pode-se obeservar que o produto interno do numerador terá valor quase nulo, fazendo com que os β's de Polak-Ribière e de Fletcher-Reeves tenham valores praticamente idênticos, isto implicará que o parâmetro ϕ deverá receber o valor 0. O fator importante dessa estratégia é que como a variável β multiplica d_k na fórmula de atualização da direção d_{k+1}, isto faria com que a direção mudasse de sentido. Assim, a estratégia se resume:

Algoritmo 7 Estratégia Proposta 1

Caso os gradientes atual e anterior forem $\approx$ perpendiculares. $\nabla f(x_k) \perp \nabla f(x_{k-1})$.

se $\langle cos(\nabla f(x_k) \angle \nabla f(x_{k-1})) > 0.8 \rangle$ **então**
  $\phi = 0$. *Assim faz $\beta^H = \beta^{FR}$.*
fim se

Caso os gradientes atual e anterior forem $\approx$ paralelos ou não perpendicular.

se $\langle cos(\nabla f(x_k) \angle \nabla f(x_{k-1})) \leq 0.8 \rangle$ **então**
  $\phi = 1$. *Assim faz $\beta^H = \beta^{PR}$.*
fim se

5.2 O parâmetro ϕ_2

Assim como a variável β "dosa" a direção d no método, o parâmetro ϕ funciona da mesma maneira mas, regulando a variável β. Podemos gerar uma classe inteira de β's somente variando a forma da variável ϕ. Uma das formas que optamos gera uma sequência

em que seus valores são decrescentes e ficam num número considerável de vezes no intervalo [1 , 0.5).

$$\phi = \sqrt[10]{\left(\frac{1}{k}\right)^2}.$$

O capítulo anterior mostra que o β^{PR} tem um bom comportamento para os pontos iniciais mais afastados de x^*, isto faz com que a estratégia consiste em começar com o β híbrido mais próximos do β^{PR} e assim trazer os iterandos para mais próximos do minimizador. Somente quando os iterandos estivessem mais próximos de x^* fazer com que o híbrido passe a ter o valor próximo do β^{FR} para o qual podemos observar nos testes numéricos do capítulo anterior que a velocidade de convergência era superior para pontos próximos do minimizador.

Capítulo 6

Problemas Testes de Otimização Irrestrita - β^H

6.1 Testes Numéricos - ϕ_1

As tabelas a seguir exibem os resultados obtidos com o parâmetro ϕ_1.

6.1.1 Funções Quadráticas

Na função quadrática já esperávamos os resultados da Tabela 6.1, uma vez que sabemos que para qualquer β utilizado no método obteremos a convergência e em no máximo n iterações.

Problema 1: Quadrática

Quadrática 1

Quadrática 1 - ϕ_1			
Dimensão	2	12	36
Beta	FR	PR	H
Ponto que converge	$x_1 = 0$; $x_2 = 0$	$x_j = 0$	$x_j = 0$
Número iterações	2	2	2
Norma	0	0	0

Tabela 6.1: Quadrática dim=(2/12/36) ; $x_j^0 = (x_{2j-1}^0 = 10 \; ; \; x_{2j}^0 = 3)$

Quadrática 2

I) $A = H := diag(m, m-1, \cdots, 1)$; $b = (1, \cdots, 1)^T$; $x^* = (0, \cdots, 0)^T$;

Quadrática 2 - Problema I - ϕ_1			
	β^{FR}	β^{PR}	β^H
x^0	$x_j^0 = 1$		
Dimensão 10	D	44.325	41.760
Dimensão 100	D	42.661	41.686
Dimensão 1000	D	D	D
x^0	$x_j^0 = 0.2$		
Dimensão 10	D	44.051	40.397
Dimensão 100	D	44.485	44.112
Dimensão 1000	D	D	D

Tabela 6.2: Quadrática 2; Problema I; Busca Linear: Armijo

II) $A = H := diag(10m, 5m, m, m-1, \cdots, 1)$; $b = (1, \cdots, 1)^T$; $x^* = (0, \cdots; 0)^T$;

Quadrática 2 - Problema II- ϕ_1			
	β^{FR}	β^{PR}	β^H
x^0	$x_j^0 = 1$		
Dimensão 10	D	40.532	45.954
Dimensão 100	D	44.438	42.166
Dimensão 1000	D	D	D
x^0	$x_j^0 = 0.2$		
Dimensão 10	726	42.675	43.749
Dimensão 100	D	43.559	43.523
Dimensão 1000	D	D	D

Tabela 6.3: Quadrática 2; Problema II; Busca Linear: Armijo

6.1.2 Função Rosenbrock

Nesta função podemos mencionar que houve resultados muito expressivos. Quando escolhemos chutes iniciais afastados do que chamamos de x^*, tanto para dimensão pequena quanto grande o desempenho do método com este β foi muito competitivo. Apesar das dimensões 12 e 36 o número de iterações serem ligeiramente maiores, podemos ainda dizer que seus resultados foram satisfatórios visto que na dimensão 2, o problema obtém uma solução excelente para um número de iterações pequeno.

Problema 2: Rosenbrock

Dimensão: $m = n$
Funções: $f_{2i-1}(x) = 10(x_{2i} - x_{2i-1}^2)$
$f_{2i}(x) = 1 - x_{2i-1} \quad \forall i = 1, \cdots, m/2$
Ponto inicial: $x^0 = (x_j^0)$ onde $x_{2j-1}^0 = -1, 2$; $x_{2j}^0 = 1 \quad \forall j = 1, \cdots, n/2$
Minimizador: $f = 0$ com $(1; \ldots; 1)$

N° de iterações - Rosenbrock - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 2	D	215.554*	D
Dimensão 12	D	19.231	D
Dimensão 36	D	19.252	D

Tabela 6.4: Rosenbrock dim=2/12/36 ; $x^0 = (-1,2 \;\; ; \;\; 1,0)$

N° de iterações - Rosenbrock - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 2	1.503	194.980*	1.514
Dimensão 12	1.644	1.156	1.652
Dimensão 36	1.723	1.215	1.731

Tabela 6.5: Rosenbrock dim=2/12/36 ; $x^0 = (0,7 \;\; ; \;\; 0,7)$

6.1.3 Função Freudenstein-Roth

O problema Freudenstein-Roth assim como o problema Himmelblau que expomos mais a frente é sem dúvida os mais importantes para os nossos testes. Nestes, existem pelo menos dois minimizadores que possuem diferentes imagens. Quando entramos com um chute inicial afastado de x^* o β^{FR} converge para um minimizador local e o β^{PR} para o minimizador global mas, com 371 iterações enquanto que com o β^{LL} converge para o minimizador global somente com 104 iterações. A eficiência e robustez do β^{LL} também ocorre quando fazemos chute inicial próximo de x^*. Mais uma vez os resultados foram promissores ao utilizarmos o β proposto neste trabalho e apresentamos nas Tabelas 6.6 e 6.6.

Problema 3: Freudenstein-Roth

Dimensão: $n = 2$ e $m = 2$
Funções: $f_1(x) = -13 + x_1 + ((5 - x_2)x_2 - 2)x_2$
$f_2(x) = -29 + x_1 + ((x_2 + 1)x_2 - 14)x_2$
Ponto inicial: $x^0 = (0,5 \; ; \; -2)$
Minimizador: $f = 0$ com $(5; 4)$

N° de iterações - Freudenstein-Roth - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 2	NC^*	371	104

Tabela 6.6: Freudenstein-Roth; $x^0 = (0,5 \ ; \ -2,0)$

$*$ Aqui dissemos que o teste para o β^{FR} NC (não converge), pois o algoritmo convergiu para o minimizador *local* e não para o minimizador *global* como os outros β's.

N° de iterações - Freudenstein-Roth - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 2	54	459	122

Tabela 6.7: Freudenstein-Roth; $x^0 = (4,5 \ ; \ 3,5)$

6.1.4 Função Powell

Na função Powell assim como o β^{FR} e β^{PR} diverge ou não converge para chute inicial afastado de x^* nas dimensões testadas o β^H também diverge. Os resultados obtidos com o β^H são competitivos e podem ser conferidos nas tabelas seguintes.

Problema 4: Powell-Extended-Singular

Dimensão: n =múltiplo de 4 e $m = n$
Funções:
$$f_{4i-3}(x) = x_{4i-3} + 10x_{4i-2}$$
$$f_{4i-2}(x) = \sqrt{5}(x_{4i-1} - x_{4i})$$
$$f_{4i-1}(x) = (x_{4i-2} - 2x_{4i-1})^2$$
$$f_{4i}(x) = \sqrt{10}(x_{4i-3} - x_{4i})^2$$
Ponto inicial: $x^0 = (x_j^0)$ onde $x_{4j-3}^0 = 3$; $x_{4j-2}^0 = -1$; $x_{4j-1}^0 = 0$; $x_{4j}^0 = 1$
Minimizador: $f = 0$ com $(0; \ldots; 0)$

N° de iterações - Powell-Extended-Singular - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 4	D	NC	D
Dimensão 12	D	NC	D
Dimensão 36	D	NC	D

Tabela 6.8: Powell dim=4/12/36 ; $x_j^0 = (x_{4j-3}^0 = 3\,; x_{4j-2}^0 = -1\,; x_{4j-1}^0 = 0\,; x_{4j}^0 = 1)$

N° de iterações - Powell-Extended-Singular - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 4	203	3.960	212
Dimensão 12	221	4.300	229
Dimensão 36	1.303	NC	1.323

Tabela 6.9: Powell dim=4/12/36 ; $x_j^0 = (x_{4j-3}^0 = 0,01\,; x_{4j-2}^0 = 0\,; x_{4j-1}^0 = 0,01\,; x_{4j}^0 = 0)$

6.1.5 Função Broyden Tridiagonal

Este é outro problema muito interessante para uma análise. A função Broyden Tridiagonal mostrou um excelente desempenho com o β^H. Logo na dimensão 3 os resultados obtidos foram muito bons, tanto no chute inicial afastado de x^* quanto perto. Obtivemos para o chute inicial longe 323 iterações com o β^H contra 304 e 6.327, respectivamente β^{FR} e β^{PR}. Sendo que para o chute inicial mais próximo de x^* o β^{FR} sequer convergiu enquanto que obtivemos 9.592 iterações tanto para β^{PR} como para o β^H proposto. Os resultados são também bastantes expressivos nas dimenões 12 e 36, com divergência em maior parte ao utlizarmos β^{FR} e não convergência por parte do β^{PR}, mostramos uma grande eficiência do β^H.

Problema 5: Broyden Tridiagonal

Dimensão: $n \geq 3$ e $m = n$
Funções: $f_i(x) = (3 - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1$
onde $x_0 = x_{n+1} = 0$
Ponto inicial: $x^0 = (-1; \ldots; -1)$
Minimizador: $f = 0$

N° de iterações - Broyden Tridiagonal - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 3	304	6.327	323
Dimensão 12	341	7.250	350
Dimensão 36	D	23.146	D

Tabela 6.10: Broyden dim=3/12/36 ; $x^0 = (-1\,; -1\,; -1)$

N° de iterações - Broyden Tridiagonal - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 3	D	9.592	9.592
Dimensão 12	D	NC	49.900
Dimensão 36	D	NC	NC

Tabela 6.11: Broyden dim=3/12/36 ; $x_j^0 = (x_{3j-2}^0 = -2\,;\ x_{3j-1}^0 = 1\,;\ x_{3j}^0 = -0,5)$

6.1.6 Função Himmelblau

Quando incluímos a variável β que propomos ao método e testamos nesta última função, os resultados obtidos são bastantes satisfatórios. Se analisarmos as tabelas apresentadas ao longo deste trabalho para este mesmo problema, veremos que quando utilizamos o β proposto, este consegue competir de forma que atende os bons requesitos de um β (obtendo convergência para o minimizador global tanto quando adotamos um chute inicial longe quanto perto).

Problema 6

Dimensão:	$n = 2$ e $m = 2$
Funções:	$f_1(x) = x_1^2 + x_2 - 11$
	$f_2(x) = x_1 + x_2^2 - 7$
Ponto inicial:	$x^0 = (0\ ;\ 0)$
Minimizador:	$f = 0$ com $(3; 2)$

Nº de iterações - Função Himmelblau - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 2	NC	31	31

Tabela 6.12: Função Himmelblau dim=2 ; $x^0 = (0,0\ \ ;\ \ 0,0)$

Nº de iterações - Função Himmelblau - ϕ_1			
	β^{FR}	β^{PR}	β^H
Dimensão 2	28	27	24

Tabela 6.13: Função Himmelblau dim=2 ; $x^0 = (2,5\ \ ;\ \ 1,5)$

6.2 *Performance Profile*

Apesar da forma quase unânime em que o β^H se mostrou satisfatório nas funções utilizadas apresentadas nas tabelas, podemos demonstrar ainda mais tais resultados com um estudo da performance do mesmo.

Chamamos de *Performance Profile* a ação de medir o desempenho de um método, algoritmo, aplicação, etc, e através deste resultado podemos ponderar a eficiência desses objetos de análise (ver [3]).

Podemos dizer que o *performance profile* nos oferece uma análise comparativa completa que fazemos entre os β's, métodos, algoritmos, aplicações, etc, pois ele compara o desempenho de n_b estratégias de um conjunto B ao resolver n_p problemas do conjunto P, em relação a alguma medida de desempenho (tal como o número de iterações, tempo de

execução, etc) de modo a escolher aquele em que o cumprimento satisfatório na resolução de uma classe de problemas seja o melhor. O valor $num.iter_{p,b}$ é o valor dessa medida para o problema p, quando este é resolvido pelo método dos Gradientes Conjugados com a estratégia b.

Definimos uma razão $r_{p,b}$, para cada problema p e estratégia b, dada por

$$r_{p,b} = \frac{num.iter_{p,b}}{\min\{num.iter_{p,b} : b \in B\}}.$$

Assumimos $r_M \geq r_{p,b}$ tal que p, b escolhido, e $r_{p,b} = r_M$, se somente se, a estratégia b não resolve o problema p, sendo que r_M é um valor que não afete o desempenho (escolhemos um valor muito grande).

A razão definida anteriormente mostra o efeito do β na resolução de um determinado problema, todavia, não apresenta a avaliação geral do desempenho dos β's e assim definimos a função que apresenta tal *perfil de desempenho*,

$$\rho_b(\tau) = \frac{1}{n_p}\text{tamanho}\{p \in P : r_{p,b} \leq \tau\}.$$

Para cada valor de τ, os valores de ρ equivalem ao percentual de problemas resolvidos pela estratégia b com resultado menor ou igual a τ multiplicado pelo valor da medida mínima.

Sendo assim, as imagens de ρ quando utilizamos τ igual a 1 (um), correspondem à eficiência das estratégias e para o valor de τ máximo, temos a robustez dos mesmos.

O processo de criação de um *performance profile* não é tão complexo. Este, gera uma tabela com dados dos resultados obtidos do método com os betas utilizados aplicados aos problemas que usamos como testes e aplica alguns conceitos. Para enxegarmos melhor a ideia do *performance profile* podemos supor didaticamente que vamos assumir a medida de

análise como sendo o número de iterações (assim como já foi dito que poderíamos analisar qualquer medida comparativa como: tempo de execução, número de iterações, etc).

- Então o primeiro "passo que damos" é criar uma tabela (matriz) "t" com os dados que obtivemos (números de iterações encontrados com cada beta para cada problema).

Matriz "t"			
b (betas) / p (problemas)	β^{FR}	β^{PR}	β^{H}
Problema 1	1.300	NaN	12
Problema 2	100.000	15	110
⋮	⋮	⋮	⋮
Problema 30	140	1.125	135

Tabela 6.14: Valores e Problemas Fictícios - matriz t

$*num.iter_{p,b} = 1.300, NaN, 12, 100.000$, etc.

- O segundo passo é criar uma matriz "r" em que os valores desta matriz serão os valores da matriz "t" divididos pelo menor valor encontrado nas suas respectivas linhas,

$$r_{p,b} = \frac{num.iter_{p,b}}{\min\{num.iter_{p,b} : b \in B\}}.$$

Matriz "r"			
b (betas) / p (problemas)	β^{FR}	β^{PR}	β^{H}
Problema 1	108,33 (1.300/12)	NaN (NaN/12)	1 (12/12)
Problema 2	6.666,66 (100.000/15)	1 (15/15)	7,33 (110/15)
⋮	⋮	⋮	⋮
Problema 30	1,037 (140/135)	8,33 (1.125/135)	1 (135/135)

Tabela 6.15: Valores e Problemas Fictícios - matriz r

Linha 1	$\min\{num.iter_{p,b} : b \in B\} = 12$
Linha 2	$\min\{num.iter_{p,b} : b \in B\} = 15$
⋮	⋮

Tabela 6.16: Valores e Problemas Fictícios - mínimo por linha.

- O próximo passo gera uma função de distribuição acumulada para a razão de desempenho, ou seja, cria um função densidade que exibe o resultado da análise (robustez e eficiência), mostrando a probabilidade de que a razão de desempenho associada ao método esteja dentro de um fator τ do melhor desempenho obtido:

$$\rho_b(\tau) = \frac{1}{n_p}\text{tamanho}\{p \in P : r_{p,b} \leq \tau\}.$$

  Para instituir esta função, o algoritmo inicia criando um intervalo $\tau = [1, max(r_{p,b})]$ (que cresce/diminue de 0.1) para compor o eixo das abscissas. Em seguida o processo começa a determinar (na coluna) a quantidade (tamanho) de problemas que possuem a taxa $r_{p,b}$ com valores menores ou iguais a um determinado τ e faz o quociente deste tamanho encontrado pelo número de problemas n_p. Este processo é feito para todas as colunas (β's). A partir disso, é plotado o percentual de problemas resolvidos para cada τ.

Um código implementado em *Matlab* que gera o *performance profile* segue.

```
function perf(T,logplot)
%PERF    Performace profiles
%
% PERF(T,logplot)-- produces a performace profile as described in
%   Benchmarking optimization software with performance profiles,
%   E.D. Dolan and J.J. More',
%   Mathematical Programming, 91 (2002), 201--213.
% Each column of the matrix T defines the performance data for a solver.
% Failures on a given problem are represented by a NaN.
% The optional argument logplot is used to produce a
% log (base 2) performance plot.
%
% This function is based on the perl script of Liz Dolan.
%
% Jorge J. More', June 2004

if (nargin < 2) logplot = 0; end

colors  = ['m' 'b' 'r' 'g' 'c' 'k' 'y'];
lines   = [':' '-' '-.' '--'];
markers = ['x' '*' 's' 'd' 'v' '^' 'o'];

[np,ns] = size(T);

% Minimal performance per solver
```

```
minperf = min(T,[],2);

% Compute ratios and divide by smallest element in each row.

r = zeros(np,ns);
for p = 1: np
  r(p,:) = T(p,:)/minperf(p);
end

if (logplot) r = log2(r); end

max_ratio = max(max(r));

% Replace all NaN's with twice the max_ratio and sort.

r(find(isnan(r))) = 2*max_ratio;
r = sort(r);

% Plot stair graphs with markers.

clf;
for s = 1: ns
 [xs,ys] = stairs(r(:,s),[1:np]/np);
 option = ['-' colors(s) markers(s)];
 plot(xs,ys,option,'MarkerSize',3);
 hold on;
```

```
end

% Axis properties are set so that failures are not shown,
% but with the max_ratio data points shown. This highlights
% the "flatline" effect.

axis([ 1 1.1*max_ratio 0 1 ]);

% Legends and title should be added.
```

Os perfis de desempenhos seguintes exibem os resultados dos testes com o método dos Gradientes Conjugados mostrando a atuação dos três β's estudados, adotando a primeira estratégia do parâmetro ϕ.

Fizemos uma divisão dos problemas estudados para construção de seus *performance profiles* através de suas dimensões. Separamos os problemas testes em dimensões pequenas, médias e grandes, considerando assim os problemas com dimensões 2, 3 ou 4 como sendo de dimensões pequenas, os problemas com dimensão 12 como sendo problemas médios e problemas com dimensão 36 sendo problemas grandes e geramos os seus respectivos *performance profiles*. Considerando além das dimensões, construímos os gráficos de desempenhos considerando também o chute inicial adotado e portanto para cada divisão há dois *performance profiles*: um para o chute inicial próximo do minimizador x^* e outro gráfico de desempenho para chute inicial longe de x^*.

Legenda:

vermelho-β^H, azul-β^{PR} e lilás-β^{FR}.

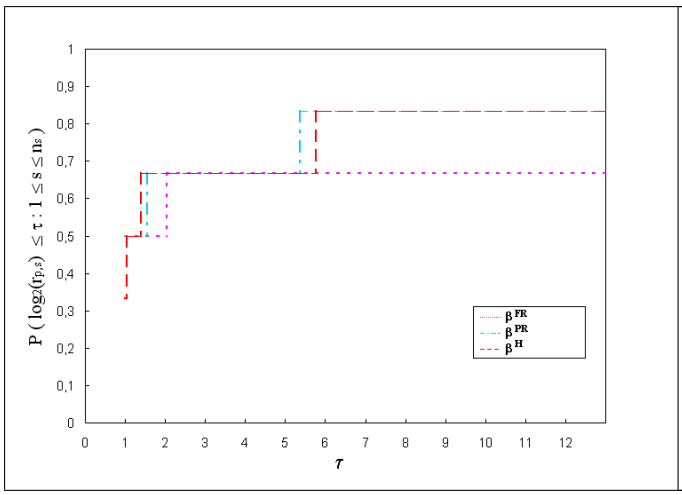

Figura 6.1: x^0 *longe com dimensão pequena*

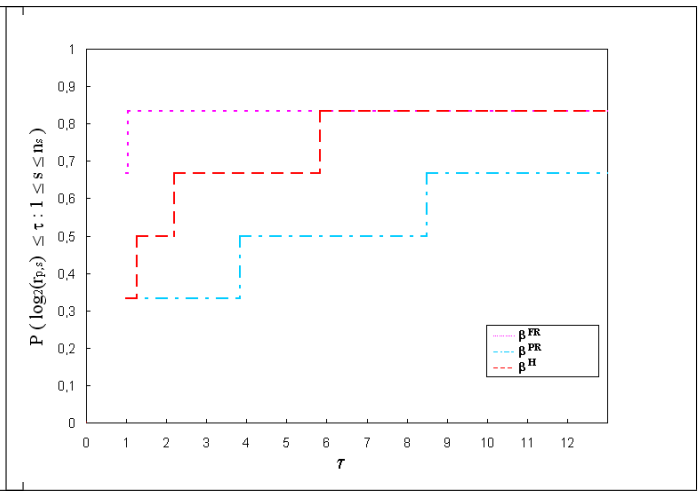

Figura 6.2: x^0 *perto com dimensão pequena*

Os primeiros perfis de desempenho que construímos foram obtidos com os dados dos resultados dos problemas com a dimensão pequena. A análise que podemos fazer com estes foi que para chute inicial mais "longe" de x^* (Figura 6.1) tanto em questão de eficiência quanto de robustez o β híbrido obteve melhor desempenho. Esta afirmação é feita pois se analisarmos no *performance profile* pode-se observar que a curva de cor vermelha (β^H) inicia com altura superior às outras curvas (azul-β^{PR} e lilás-β^{FR}) mostrando sua maior eficiência. E ao final de τ (eixo das abscissas) a curva que alcançou também maior altura é aquele β que possui melhor robustez (neste caso, todos eles), ou seja, todos os β's conseguiram resolver grande parte dos problemas testados (levando em consideração somente aqueles de dimensão pequena - 2, 3 ou 4) e que no performance é mostrado que as curvas alcançam percentual próximo a 1. Já quando entramos com chute inicial "perto" de x^* (Figura 6.2) o β^{FR} obtém melhor eficiência, pois este inicia em aproximadamente 0,82 (eixo das ordenadas) enquanto que β^H inicia com 0,35 contra 0,18 do β^{PR}. E ao final de τ em 0,52, todos também conseguiram resolver parte dos problemas testados chegando a conclusão da robustez desses β's.

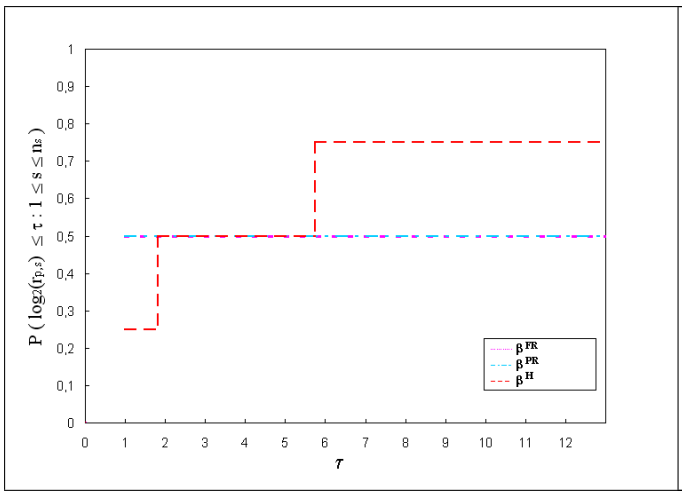

Figura 6.3: x^0 *longe com dimensão média*

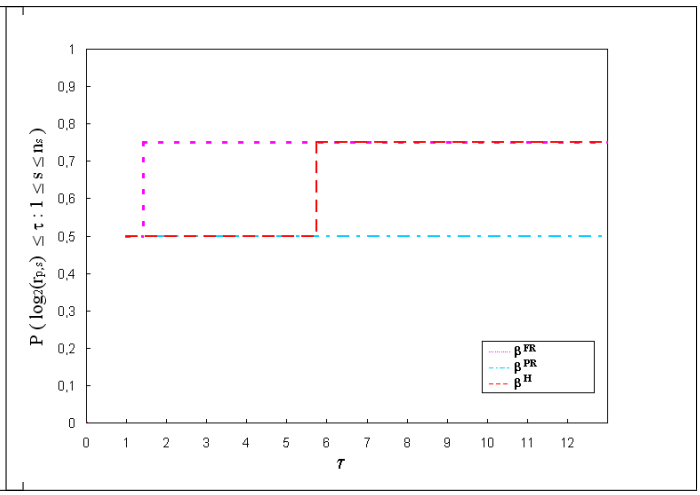

Figura 6.4: x^0 *perto com dimensão média*

Os *performances profiles* (Figuras 6.3 e 6.4) são os perfis de desempenho com os dados dos resultados dos problemas com dimensão média. Podemos perceber na Figura 6.4 que quando entramos com chute inicial perto do minimizador de f o β^{PR} (curva azul) inicia em 0,5 contra 0,25 de β^H e β^{FR} e assim constata a melhor eficiência de β^{PR}. E novamente, todos os β's obtêm uma robustez satisfatória, pois ao final de tau estes conseguem alcançar percentual considerável.

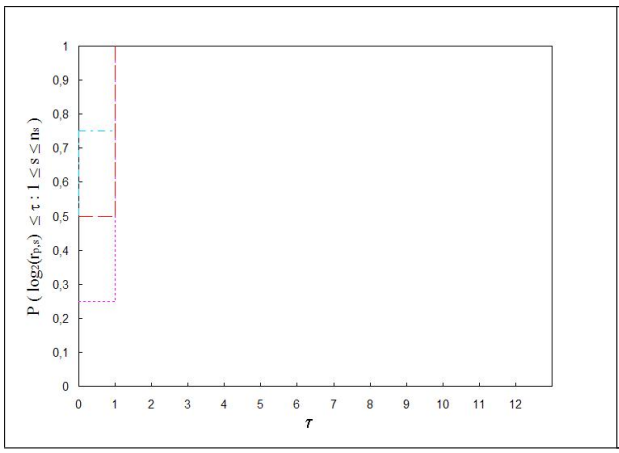

Figura 6.5: x^0 *longe com dimensão grande*

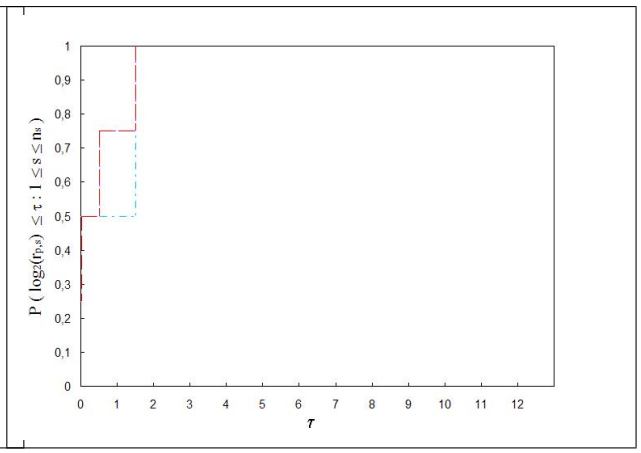

Figura 6.6: x^0 *perto com dimensão grande*

Além de elaborarmos *performance profiles* que só levavam em consideração a dimensão pequena, média e grande, ou o chute inicial, foi também construída uma Tabela (6.17) contendo todos os resultados encontrados em nossas análises para gerarmos um *performance profile* geral dos nossos dados. Cada problema com um ponto inicial diferente corresponde a uma nova linha na matriz, por exemplo, para o problema Rosenbrock com dimensão 2, geramos um *performance profile* tanto quando adotamos chute inicial perto quanto para chute inicial longe. Portanto, este gráfico de desempenho geral incluirá todos os problemas testes admitindo serem diferentes não só pelo problema em si, mas também por suas dimensões e o chute inicial adotado.

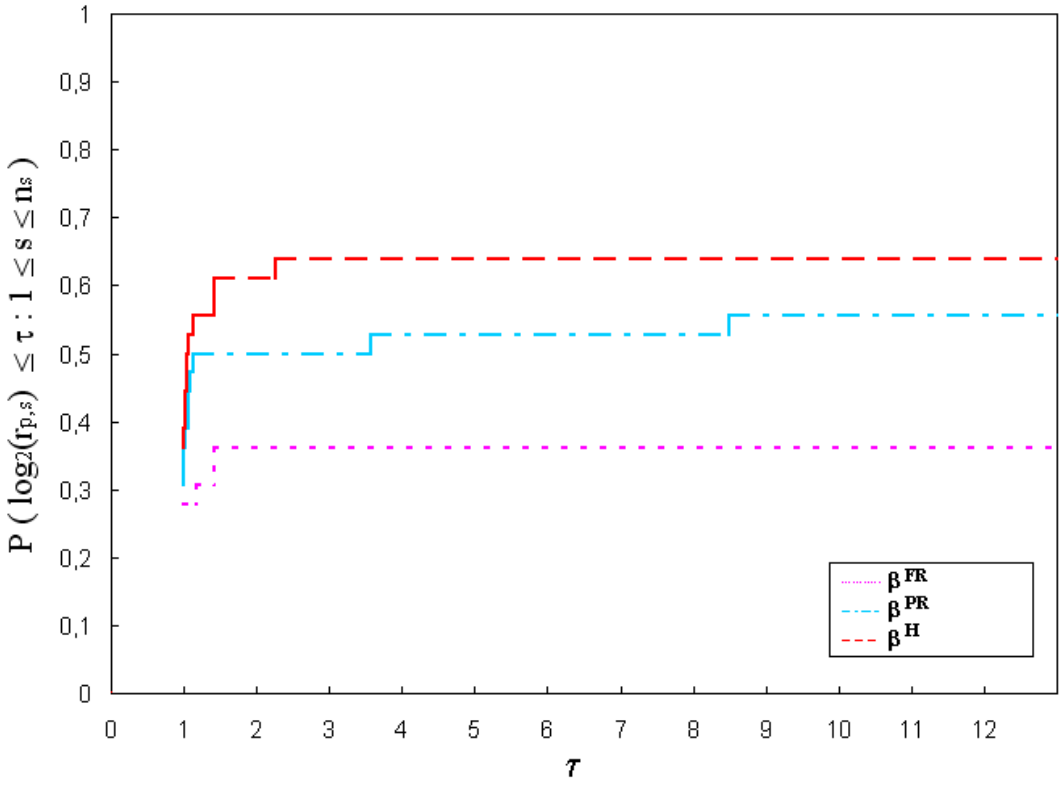

Figura 6.7: *Performance Profile Geral*

Todos os Resultados Obtidos			
b (betas) / p (problemas)	β^{FR}	β^{PR}	β^{H}
Problema Quadrático I CIL	2	2	2
Problema Quadrático I CIP	2	2	2
Problema Quadrático II A Dim10 CIL	D	44.325	41.760
Problema Quadrátcio II A Dim100 CIL	D	42.661	41.686
Problema Quadrático II A Dim1000 CIL	D	D	D
Problema Quadrático II A Dim10 CIP	D	44.051	40.397
Problema Quadrático II A Dim100 CIP	D	44.485	44.112
Problema Quadrático II A Dim1000 CIP	D	D	D
Problema Quadrático II B Dim10 CIL	D	40.532	45.954
Problema Quadrático II B Dim100 CIL	D	44.438	42.166
Problema Quadrático II B Dim1000 CIL	D	D	D
Problema Quadrático II B Dim10 CIP	726	42.675	43.749
Problema Quadrático II B Dim100 CIP	D	43.559	43.523
Problema Quadrático II B Dim1000 CIP	D	D	D
Problema Rosenbrock Dim2 CIL	D	215.554	D
Problema Rosenbrock Dim12 CIL	D	19.231	D
Problema Rosenbrock Dim36 CIL	D	19.252	D
Problema Rosenbrock Dim2 CIP	1.503	194.980	1.514
Problema Rosenbrock Dim12 CIP	1.644	1.156	1.652
Problema Rosenbrock Dim36 CIP	1.723	1.215	1.731
Problema Freudenstein-Roth CIL	D	371	104
Problema Freudenstein-Roth CIP	54	459	122
Problema Powell Dim4 CIL	D	NC	D
Problema Powell Dim12 CIL	D	NC	D
Problema Powell Dim36 CIL	D	NC	D
Problema Powell Dim4 CIP	203	3.960	212
Problema Powell Dim12 CIP	221	4.300	229
Problema Powell Dim36 CIP	1.303	NC	1.323
Problema Broyden Dim3 CIL	304	6.327	323
Problema Broyden Dim12 CIL	341	7.250	350
Problema Broyden Dim36 CIL	D	23.146	D
Problema Broyden Dim3 CIP	D	9.592	9.592
Problema Broyden Dim12 CIP	D	NC	49.000
Problema Broyden Dim36 CIP	D	NC	NC
Problema Himmelblau CIL	D	31	31
Problema Himmelblau CIP	28	27	24
Dim:dimensão. CIL:chute inicial longe. CIP: chute inicial perto			

Tabela 6.17: Número de Iterações

6.2.1 Robustez

Uma das características do *Performance Profile* é ser capaz de avaliar o desempenho tal como a robustez dos métodos ou, no nosso caso, os β'*s* assim estudados.

Quando analisamos a robustez do β proposto, podemos destacar que neste aspecto ele foi, em muitos casos, o que obteve melhor desempenho ou então se saiu tão bem quanto os demais (com exceção em dimensões grandes quando o chute inicial é dado próximo do ótimo).

6.2.2 Eficiência

Ao analisarmos a eficiência do β^H podemos mencionar que não se obteve um resultado tão bom quanto a sua robustez. Ele, em geral, ou resolve os problemas testados em maior número de iterações – se comparado aos outros β'*s* – ou se saiu equiparado a eles, podendo este aspecto ser visto em todos os *perfis de desempenho*, ou seja, a imagem do β^H é iniciado ou abaixo daquele β em que se teve o melhor desempenho ou sobreposto (igual desempenho) a tal β.

6.3 Testes Numéricos - ϕ_2

Também foram realizados testes numéricos para o método dos Gradientes Conjugados utilizando a estratégia do parâmetro ϕ_2 no cálculo do β^H. Foram resolvidos os mesmo problemas testes já apresentados nas seções anteriores deste capítulo, obtendo os resultados a seguir:

Quadrática

Quadrática - ϕ_2			
	β^{FR}	β^{PR}	β^{H}
Dimensão 2	2	2	2
Dimensão 12	2	2	2
Dimensão 36	2	2	2

Tabela 6.18: Quadrática dim=(2/12/36) ; $x_j^0 = (x_{2j-1}^0 = 10 \; ; \; x_{2j}^0 = 3)$

Rosenbrock

N° de iterações - Rosenbrock - ϕ_2			
	β^{FR}	β^{PR}	β^{H}
Dimensão 2	D	NC	D
Dimensão 12	D	19.231	D
Dimensão 36	D	19.252	D

Tabela 6.19: Rosenbrock dim=2/12/36 ; $x^0 = (-1,2 \quad ; \quad 1,0)$

N° de iterações - Rosenbrock - ϕ_2			
	β^{FR}	β^{PR}	β^{H}
Dimensão 2	1.503	NC	1.519
Dimensão 12	1.644	1.156	1.646
Dimensão 36	1.723	1.215	1.725

Tabela 6.20: Rosenbrock dim=2/12/36 ; $x^0 = (0,7 \quad ; \quad 0,7)$

Freudenstein-Roth

N° de iterações - Freudenstein-Roth - ϕ_2			
	β^{FR}	β^{PR}	β^{H}
Dimensão 2	NC	371	NC

Tabela 6.21: Freudenstein-Roth; $x^0 = (0,5 \quad ; \quad -2,0)$

N° de iterações - Freudenstein-Roth - ϕ_2			
	β^{FR}	β^{PR}	β^{H}
Dimensão 2	54	459	148

Tabela 6.22: Freudenstein-Roth; $x^0 = (4,5 \quad ; \quad 3,5)$

Função Powell

N° de iterações - Powell-Extended-Singular - ϕ_2			
	β^{FR}	β^{PR}	β^H
Dimensão 4	D	NC	D
Dimensão 12	D	NC	D
Dimensão 36	D	NC	D

Tabela 6.23: Powell dim=4/12/36 ; $x_j^0 = (x_{4j-3}^0 = 3\,; x_{4j-2}^0 = -1\,; x_{4j-1}^0 = 0\,; x_{4j}^0 = 1)$

N° de iterações - Powell-Extended-Singular - ϕ_2			
	β^{FR}	β^{PR}	β^H
Dimensão 4	203	3.960	204
Dimensão 12	221	4.300	222
Dimensão 36	1.303	NC	1.329

Tabela 6.24: Powell dim=4/12/36 ; $x_j^0 = (x_{4j-3}^0 = 0,01\,; x_{4j-2}^0 = 0\,; x_{4j-1}^0 = 0,01\,; x_{4j}^0 = 0)$

Broyden Tridiagonal

N° de iterações - Broyden Tridiagonal - ϕ_2			
	β^{FR}	β^{PR}	β^{H}
Dimensão 3	304	6.327	315
Dimensão 12	341	7.250	342
Dimensão 36	D	23.146	D

Tabela 6.25: Broyden dim=3/12/36 ; $x^0 = (-1\,;-1\,;-1)$

N° de iterações - Broyden Tridiagonal - ϕ_2			
	β^{FR}	β^{PR}	β^{H}
Dimensão 3	D	9.592	D
Dimensão 12	D	NC	D
Dimensão 36	D	NC	D

Tabela 6.26: Broyden dim=3/12/36 ; $x_j^0 = (x_{3j-2}^0 = -2\,;\; x_{3j-1}^0 = 1\,;\; x_{3j}^0 = -0,5)$

Himmelblau

N° de iterações - Função Himmelblau - ϕ_2			
	β^{FR}	β^{PR}	β^H
Dimensão 2	NC	31	NC

Tabela 6.27: Função Himmelblau dim=2 ; $x^0 = (0,0 \quad ; \quad 0,0)$

N° de iterações - Função Himmelblau - ϕ_2			
	β^{FR}	β^{PR}	β^H
Dimensão 2	28	27	46

Tabela 6.28: Função Himmelblau dim=2 ; $x^0 = (2,5 \quad ; \quad 1,5)$

Podemos perceber que os resultados dos testes numéricos implementados com o parâmetro ϕ_2 também são muito promissores. Apesar de no problema Freudenstein-Roth não conseguir uma convergência para o minimizador global, obteve convergência para o minimizador local. Se analisarmos o seu desempenho de um modo geral, pode-se dizer que seu papel no controle da variável β apresentou-se competitividade em relação aos β^{FR} e β^{PR}.

Ao compararmos o desempenho dos parâmetro ϕ_1 e ϕ_2 constatamos que o parâmetro ϕ_1 teve um melhor comportamento sobre o parâmetro ϕ_2 e por isso os resultados com a estratégia do parâmetro ϕ_2, embora tenham sido exaustivamente testados, não foram expostos ou analisados de forma extensa no texto da dissertação.

Conclusão

O método dos Gradientes Conjugados ao longo dos anos passou por algumas modificações consideráveis. No seu início era determinado a resolver problemas quadráticos e por isso era comumente utilizado para buscar soluções de sistemas lineares. Contudo, percebeu-se que com sua estrutura era possível trabalhar com uma vasta classe de problemas (respeitando algumas hipóteses) e assim permitindo resolver problemas não lineares diversos. Os problemas solucionados pelo método dos Gradientes Conjugados são resolvidos de forma iterativa fazendo uma atualização no ponto atual $x_{k+1} = x_k + \alpha d_k$.

Neste trabalho, estudamos duas estratégias para a atualização do parâmetro β_k clássicas, β Fletcher-Reeves e β Polak-Ribière, amplamente utilizadas na literatura.

Baseados nos desempenhos apresentados por essas estratégias durante os testes computacionais, propomos um β híbrido que combina os β^{FR} e β^{PR} em sua fórmula, por meio de um coeficiente de controle ϕ.

Foram apresentadas e testadas numericamente duas estratégias para a atualização desse coeficiente de controle, ϕ_1 e ϕ_2, que buscavam combinar as boas características apresentadas pelos β^{FR} e β^{PR}.

Diantes dos testes numéricos realizados, pode-se observar que o β híbrido mostrou-se competitivo em questão de robustez e eficiência em relação aos dois β's clássicos, tendo a estratégia de controle ϕ_1 melhor desempenho que a estratégia de controle ϕ_2.

Para trabalhos futuros pretende-se testar novas estratégias para ϕ e implementar

novas funções pretendendo assim aumentar o número de funções estudadas com o Método dos Gradientes Conjugados.

Referências Bibliográficas

[1] HESTENES, M. R.; STIEFEL, E. L. Methods of conjugate gradients for solving linear systems. *Journal of Research - National Bureau of Standards*, 1952.

[2] FLETCHER, R.; REEVES, C. Function minimization by conjugate gradients. *Electronic Computing Laboratory*, 1964.

[3] DOLAN, E. D.; MORÉ, J. J. Benchmarking optimization software with performance profiles. *Math. Program. Ser.*, 2002.

[4] TEIXEIRA, F. C. A. *Aproximações Arbitrárias de Módulo e Fase na Modelagem de Sistemas Lineares Invariantes ao Deslocamento com Técnicas de Otimização "Soft-Computing"*. Dissertação (Mestrado) — Universidade de Brasília, 2008.

[5] CHONG, E. K. P.; ZAK, S. H. *An Introduction to Optimization*. [S.l.]: Wiley-Interscience, 1996.

[6] POLAK, E.; RIBIÈRE, G. Note sur la convergence de methodes de directions conjugee. *Rev. Fr. Inform. Rech. Oper.*, v. 16-R1, p. 35–43, 1969.

[7] LUENBERGER, D. G. *Introduction to Linear and Nonlinear Programming*. 4th. ed. Boston: McGraw-Hill, 1973.

[8] NOCEDAL, J.; WRIGHT, S. *Numerical Optimization*. [S.l.]: Springer Series on Operations Research, 1999.

[9] IZMAILOV, A.; SOLODOV, M. *Otimização - Métodos Computacionais*. [S.l.: s.n.], 2007.

[10] VIEIRA, L. C. L. M. *Estudo de Algoritmos de Integração Elemento por Elemento para Análise Dinâmica Não Linear de Estruturas*. Dissertação (Mestrado) — Programa de Pós-Graduação em Engenharia Civil, 2004.

[11] HAYKIN, S. *Neural Networks: A Comprehensive Foundation*. [S.l.]: Macmillan, 1993.

[12] BROWN, M. *Conjugate Gradient Total Least-Squares in Geophysical Optimization Problems*. [S.l.], 2002.

[13] VELHO, H. F. C. Introdução aos problemas inversos: Aplicações em pesquisa espacial. *Instituto Nacional de Pesquisas Espaciais*, 2008.

[14] HAGER, W.; ZHANG, H. A survey of nonlinear conjugate gradient methods. *National Science Foundation under Grant*, 2005.

[15] SHEWCHUK, J. R. An introduction to the conjugate gradient method without the agonizing pain. *School of Computer Science*, 1994.

[16] BERTSEKAS, D. P. *Nonlinear Programming*. [S.l.]: Athena Scientific, 1995.

[17] HIMMELBLAU, D. M. *Applied Nonlinear Programming*. [S.l.: s.n.], 1972.

[18] DAI, Y. H.; YUAN, Y. A nonlinear conjugate gradient methods with a strong global convergence property. *Chinese National Natural Science Foundation Grants*, 2010.

[19] ZOUTENDIJK, G. Nonlinear programming, computational methods. *J. Abadie*, 1970.

[20] WOLFE, P. Convergence conditions for ascent methods. *SIAM Rev., 11*, 1969.

[21] WOLFE, P. Convergence conditions for ascent methods ii, some corrections. *SIAM Rev., 13*, 1971.

[22] FALLGREN, M. *On the Robustness of Conjugate-Gradient Methods and Quasi-Newton Methods*. Dissertação (Mestrado) — Royal Institute of Technology, 2006.

Anexo

Implementamos o algoritmo em linguagem *C* e segue abaixo.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define TAM 36

double mult_vetor_matriz(int dimensao, int numero_funcoes, double f[TAM], double J[TAM][TAM], double gradiente[TAM]){
       int i,j;

       //Zerando o vetor resultado para tirar possiveis lixos
       for(i=0;i<dimensao;i++){
                              gradiente[i]=0;
                              }
       //Multiplicando o vetor f pela matriz Jacobiana para obter o gradiente
       for(j=0;j<dimensao;j++){
            for(i=0;i<numero_funcoes;i++){
                 gradiente[j] = gradiente[j] + 2*f[i]*J[i][j];
                                    }
                                    }


 }//fim rotina multiplicacao de um vetor por uma matriz

double mult_vetores(int dimensao, double vetorA[TAM], double vetorB[TAM], double *resultado){
       int i;
      //Zerando o vetor resultado para tirar possiveis lixos
```

```
        *resultado=0;

       //Multiplicando o vetor A por um vetor B para obter um resultado
       for(i=0;i<dimensao;i++){
               *resultado = *resultado + vetorA[i]*vetorB[i];
                               }

} //fim rotina multiplicacao de vetores


double calcula_beta (int dimensao, int escolha_beta, int t, double gradiente[TAM],
                     double gradiente_anterior[TAM], double *beta ){
       int i;
       double cte_alpha;
       double beta_FR=0, beta_PR=0, beta_x=0;
       double numerador=0, numerador_FR=0, numerador_PR=0, denominador=0, denominador_FR=0, denominador_PR=0;
       double sub_g_gant[TAM];
       double resul_norma , norma_grad, norma_grad_anterior;
       double cosseno_grad_gradant;

       //calcula Beta


      if(escolha_beta==1){

            //Fletcher-Reeves
                               mult_vetores(dimensao,gradiente,gradiente,&numerador);
                               //printf("\nnumerador=%lf",numerador);
                               mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador);
                               //printf("\ndenominador=%lf",denominador);
              *beta=(numerador)/(denominador);
             //printf("\n\nbeta=%lf\n",beta);
                                  }




      if (escolha_beta==2){
```

```
     //Polak-Ribiere
                    for(i=0;i<dimensao;i++){
                                        sub_g_gant[i]=0; //Eliminar possiveis lixos
                                        sub_g_gant[i]=gradiente[i]-gradiente_anterior[i];
                                        }
                    mult_vetores(dimensao,gradiente,sub_g_gant,&numerador);
                    mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador);

              *beta=(numerador)/(denominador);
                                  }



if (escolha_beta==3){

     //Leonardo-Luziane 1

      cte_alpha=pow(1/t,0.2);

      printf("\t\t cte_alpha=%f",cte_alpha);

                          mult_vetores(dimensao,gradiente,gradiente,&numerador_FR);
                          //printf("\nnumerador=%lf",numerador);
                          mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador_FR);
                          //printf("\ndenominador=%lf",denominador);
           beta_FR=(numerador_FR)/(denominador_FR);

                    mult_vetores(dimensao,gradiente,gradiente_anterior,&numerador);
                    mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador);

            beta_x=(numerador)/(denominador);

      *beta = beta_FR - cte_alpha*beta_x;
                                } //fim beta LL1



if (escolha_beta==4){
```

```
      //Leonardo-Luziane 2

       cte_alpha=1/t;

                          mult_vetores(dimensao,gradiente,gradiente,&numerador_FR);
                          //printf("\nnumerador=%lf",numerador);
                          mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador_FR);
                          //printf("\ndenominador=%lf",denominador);
               beta_FR = (numerador_FR)/(denominador_FR);


                        for(i=0;i<dimensao;i++){
                                          sub_g_gant[i]=0; //Eliminar possiveis lixos
                                          sub_g_gant[i]=gradiente[i]-gradiente_anterior[i];
                                          }
                        mult_vetores(dimensao,gradiente,sub_g_gant,&numerador_PR);
                        mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador_PR);

                beta_PR = (numerador_PR)/(denominador_PR);

    *beta = cte_alpha*beta_PR + (1 - cte_alpha)*beta_FR;

       }



if (escolha_beta==5){

      //Leonardo-Luziane 3


      //----Calculando cosseno(grad^grad_ant)
      // cos(grad^grad_ant) = (grad . grad_ant) / (||grad||.||grad_ant||)

            //------------------numerador
          mult_vetores(dimensao,gradiente,gradiente_anterior,&numerador);

            //-----------------denominador
          mult_vetores(dimensao,gradiente,gradiente,&resul_norma);
```

```
   norma_grad=sqrt(resul_norma);

   mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&resul_norma);
   norma_grad_anterior=sqrt(resul_norma);

   denominador = norma_grad * norma_grad_anterior;
     //----------------------------

   cosseno_grad_gradant = numerador/denominador;
   printf("\n\ncosseno_grad_gradant=%f",cosseno_grad_gradant);
   printf("\n\nabs(cosseno_grad_gradant)=%f",fabs(cosseno_grad_gradant));
//-----------------------------------------

                  mult_vetores(dimensao,gradiente,gradiente,&numerador_FR);
                  //printf("\nnumerador=%lf",numerador);
                  mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador_FR);
                  //printf("\ndenominador=%lf",denominador);
      beta_FR = (numerador_FR)/(denominador_FR);


              mult_vetores(dimensao,gradiente,gradiente_anterior,&numerador);
              mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador);

       beta_x=(numerador)/(denominador);


  printf("\nt=%d",t);

  if (fabs(cosseno_grad_gradant) <= 0.8){   //(0.0 <= abs(cosseno_grad_gradant) <= 0.7)
         cte_alpha = 0;
         printf("\n F R \n");
         }
  else if( (fabs(cosseno_grad_gradant)>0.8) && (t<=10) ){
         cte_alpha = 1;
         printf("\n P R \n");
         }
  else{
         cte_alpha = 0;
         printf("\n F R \n");
```

```
            }


       *beta = beta_FR - cte_alpha*beta_x;

       }//fim LL3

if (escolha_beta==6){

     //Leonardo-Luziane 4


     //----Calculando cosseno(grad^grad_ant)
     // cos(grad^grad_ant) = (grad . grad_ant) / (||grad||.||grad_ant||)

          //------------------numerador
        mult_vetores(dimensao,gradiente,gradiente_anterior,&numerador);

          //-----------------denominador
        mult_vetores(dimensao,gradiente,gradiente,&resul_norma);
        norma_grad=sqrt(resul_norma);

        mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&resul_norma);
        norma_grad_anterior=sqrt(resul_norma);

        denominador = norma_grad * norma_grad_anterior;
          //-----------------------------

        cosseno_grad_gradant = numerador/denominador;
        printf("\n\ncosseno_grad_gradant=%f",cosseno_grad_gradant);
        printf("\n\nabs(cosseno_grad_gradant)=%f",fabs(cosseno_grad_gradant));
     //-----------------------------------------


                      mult_vetores(dimensao,gradiente,gradiente,&numerador_FR);
                      //printf("\nnumerador=%lf",numerador);
                      mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador_FR);
                      //printf("\ndenominador=%lf",denominador);
          beta_FR = (numerador_FR)/(denominador_FR);
```

```
                        mult_vetores(dimensao,gradiente,gradiente_anterior,&numerador);
                        mult_vetores(dimensao,gradiente_anterior,gradiente_anterior,&denominador);

                beta_x=(numerador)/(denominador);


             printf("\nt=%d",t);

             //cte-alpha = valor do |cosseno_grad_gradant|
             cte_alpha = fabs(cosseno_grad_gradant);

            *beta = beta_FR - cte_alpha*beta_x;

            }//fim LL4


      printf("\nescolha do beta=%d\n",escolha_beta);

} // fim rotina calcula beta


// f(x) = x^2 + y^2 + ...
double funcao_quadratica(int dimensao, int numero_funcoes, double ponto[TAM],
                  double gradiente[TAM], double *f_x){
      int i,j;
      double J[TAM][TAM];


      // Constroi a matriz identidade
      for(j=0;j<dimensao;j++){
          for(i=0;i<dimensao;i++){
                              J[i][j]=0;
                              }
                           }
      for(i=0;i<dimensao;i++){
                       J[i][i]=1;
                       }
```

```
	//gradiente
	//foi passado o vetor ponto na chamada desta funcao pois f é igual a este vetor
	mult_vetor_matriz(dimensao,numero_funcoes,ponto,J,gradiente);



	//Calcula Valor da Funcao f(x)
	*f_x=0;
	for(i=0;i<dimensao;i++){
				*f_x = *f_x + pow(ponto[i],2);
				}

} // fim rotina funcao quadratica

// f(x) = (10*(y - x^2))^2 + (1-x)^2
double funcao_rosenbrock(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM], double *f_x){
	int i,j;
	double f[TAM], J[TAM][TAM];

	//gradiente

	//f
	f[0]=10*(ponto[1]-pow(ponto[0],2));
	f[1]=1-ponto[0];

	//Jacobiana
	J[0][0]=-20*ponto[0];	J[0][1]=10;
	J[1][0]=-1;		J[1][1]=0;

	mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);



	//Calcula Valor da Funcao f(x)
	*f_x = 0;
	*f_x = pow(f[0],2) + pow(f[1],2);


} // fim funcao rosenbrock

double funcao_freudenstein_roth(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM], double *f_x){
```

```
       int i,j;
       double f[TAM], J[TAM][TAM];

      //f‘s
      f[0]=-13+ponto[0]+5*pow(ponto[1],2)-pow(ponto[1],3)-2*ponto[1];
      f[1]=-29+ponto[0]+pow(ponto[1],3)+pow(ponto[1],2)-14*ponto[1];

      //Jacobiana
      J[0][0]=1;             J[0][1]= 10*ponto[1] - 3*pow(ponto[1],2) - 2;
      J[1][0]=1;             J[1][1]= 3*pow(ponto[1],2) + 2*ponto[1] -14;

      mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);

      //Calcula valor da funcao(f(x)) no ponto
      *f_x = 0;
      *f_x = pow(f[0],2) + pow(f[1],2);

} // fim rotina funcao freudenstein-roth

double funcao_powell(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM], double *f_x){
       int i,j;
       double f[TAM], J[TAM][TAM];

       //Eliminar possiveis lixos
       for(j=0;j<dimensao;j++){
           for(i=0;i<numero_funcoes;i++){
                                       J[i][j]=0;
                                       }
                              }
        for(i=0;i<numero_funcoes;i++){
                                    f[i]=0;
                                    }

      //f‘s
      f[0]=pow(10,4)*ponto[0]*ponto[1]-1.000000000000;
      f[1]=exp(-ponto[0])+exp(-ponto[1])-1.0001000000;

      //Jacobiana
      J[0][0]=pow(10,4)*ponto[1];             J[0][1]=pow(10,4)*ponto[0];
```

```
      J[1][0]=-exp(-ponto[0]);                J[1][1]=-exp(-ponto[1]);



      for(j=0;j<dimensao;j++){
                            printf("\nf[%d]=%lf",j+1,f[j]);
                            }

      mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);

      //Calcula valor da funcao(f(x)) no ponto
      *f_x = 0;
      *f_x = pow(f[0],2) + pow(f[1],2);



} // fim funcao powell

double funcao_brown(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM]){
       int i,j;
       double f[TAM], J[TAM][TAM];

       //Eliminar possiveis lixos
       for(j=0;j<dimensao;j++){
           for(i=0;i<numero_funcoes;i++){
                                       J[i][j]=0;
                                       }
                               }
        for(i=0;i<numero_funcoes;i++){
                                     f[i]=0;
                                     }

      //F's
      f[0]=ponto[0]-1000000.000000000000000;
      f[1]=ponto[1]-(2*0.000001000000000000);
      f[2]=(ponto[0]*ponto[1])-2.00000000000000000;

      //Jacobiana
      J[0][0]=1.0000000000;                        J[0][1]=0.00000000000;
      J[1][0]=0.0000000000;                        J[1][1]=1.00000000000;
      J[2][0]=ponto[1];                            J[2][1]=ponto[0];
```

```
       //Zerando o vetor resultado para tirar possiveis lixos
       for(i=0;i<dimensao;i++){
                               gradiente[i]=0;
                               }

      //Calcula o gradiente
      mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);

} // fim rotina funcao brown

double funcao_rosenbrock_extended(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM], double *f_x){
       int i,j;
       double f[TAM], J[TAM][TAM];

       //Eliminar possiveis lixos
       for(j=0;j<dimensao;j++){
           for(i=0;i<numero_funcoes;i++){
                                        J[i][j]=0;
                                        }
                               }
        for(i=0;i<numero_funcoes;i++){
                                      f[i]=0;
                                      }



       //f
       for(i=0;i<(dimensao/2);i++){

                               f[2*i]=10*(ponto[2*i+1] - pow(ponto[2*i],2));
                               f[2*i+1]=1-ponto[2*i];
                                   }

       //Jacobiana
        //preencher a matriz com zeros
       for(j=0;j<numero_funcoes;j++){
            for(i=0;i<dimensao;i++){
                                    J[i][j]=0;
                                    }
```

```
                                }
        //Constroi a Jacobiana
        for(i=0;i<(dimensao/2);i++){
                        J[2*i][2*i]=-20*ponto[2*i];
                        J[2*i][2*i+1]=10;
                        J[2*i+1][2*i]=-1;
                                    } //fecha for

        mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);

       //Calcula valor da funcao(f(x)) no ponto
       *f_x = 0;
       for(i=0;i<dimensao;i++){
           *f_x = *f_x + pow(f[i],2);
           }

} // fim rotina funcao rosenbrock_extended


double funcao_rosenbrock_modificada(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM]){
        int i,j;
        double f[TAM], J[TAM][TAM];

        //Eliminar possiveis lixos
        for(j=0;j<dimensao;j++){
            for(i=0;i<numero_funcoes;i++){
                                        J[i][j]=0;
                                        }
                                }
         for(i=0;i<numero_funcoes;i++){
                                    f[i]=0;
                                    }


        //f
        for(i=0;i<(dimensao/2);i++){

                        if(dimensao==2){
                               f[2*i]=10*(ponto[2*i+1] - pow(ponto[i],2));
```

```
                                    }
                        else{
                                    if(i==0)
                                             f[2*i]=10*ponto[2*i+1];
                                    else
                                         f[2*i]=10*(ponto[2*i+1]-pow(ponto[i-1],2));
                               }

                        f[2*i+1]=1-ponto[2*i];
                                    }

       //Jacobiana
        //preencher a matriz com zeros
       for(j=0;j<numero_funcoes;j++){
            for(i=0;i<dimensao;i++){
                                     J[i][j]=0;
                                     }
                                     }
       //Constroi a Jacobiana
       for(i=0;i<(dimensao/2);i++){
                    if(dimensao==2){
                                       J[2*i][i]=-20*ponto[i];
                                       }
                    else{
                          if(i>0){
                                   J[2*i][i-1]=-20*ponto[i-1];
                                   }
                              }
                    J[2*i][2*i+1]=10;
                    J[2*i+1][2*i]=-1;
                                } //fecha for

       mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);

} // fim rotina funcao rosenbrock_modificada

double funcao_powell_extended(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM], double *f_x){
       int i,j;
       double f[TAM], J[TAM][TAM];
```

```
//Eliminar possiveis lixos
for(j=0;j<dimensao;j++){
    for(i=0;i<numero_funcoes;i++){
                                    J[i][j]=0;
                                    }
                          }
 for(i=0;i<numero_funcoes;i++){
                                f[i]=0;
                                }



//f
for(i=0;i<(dimensao/4);i++){
                          f[4*i]= ponto[4*i] + 10*ponto[4*i+1] ;
                          f[4*i+1]=sqrt(5)*(ponto[4*i+2] - ponto[4*i+3]) ;
                          f[4*i+2]=pow(ponto[4*i+1] - 2*ponto[4*i+2],2);
                          f[4*i+3]=sqrt(10)*pow((ponto[4*i] - ponto[4*i+3]),2);
                              }


//Jacobiana
 //preencher a matriz com zeros
for(j=0;j<numero_funcoes;j++){
     for(i=0;i<dimensao;i++){
                               J[i][j]=0;
                               }
                               }
//Constroi a Jacobiana
for(i=0;i<(dimensao/4);i++){
                J[4*i][4*i]=1;
                J[4*i][4*i+1]=10;
                J[4*i+1][4*i+2]=sqrt(5);
                J[4*i+1][4*i+3]=-sqrt(5);
                J[4*i+2][4*i+1]=2*(ponto[4*i+1] - 2*ponto[4*i+2]);
                J[4*i+2][4*i+2]=-4*(ponto[4*i+1] - 2*ponto[4*i+2]);
                J[4*i+3][4*i]=2*sqrt(10)*(ponto[4*i] - ponto[4*i]);
                J[4*i+3][4*i+3]=-2*sqrt(10)*(ponto[4*i] - ponto[4*i+3]);
                             } //fecha for Jacobiana
```

```
       mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);

      //Calcula Valor da Funcao f(x)
      *f_x = 0;
      for(i=0;i<dimensao;i++){
         *f_x = *f_x + pow(f[i],2);
         }

} // fim rotina funcao powell_extended


double funcao_broyden_tridiagonal(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM], double *f_x){
       int i,j;
       double f[TAM], J[TAM][TAM];

       //Eliminar possiveis lixos
       for(j=0;j<dimensao;j++){
           for(i=0;i<numero_funcoes;i++){
                                       J[i][j]=0;
                                       }
                              }
        for(i=0;i<numero_funcoes;i++){
                                    f[i]=0;
                                    }

       //f
       for(i=0;i<dimensao;i++){
                             if(i==0){
                                     f[i]=(3-2*ponto[i])*ponto[i] - 2*ponto[i+1] +1;
                                     }
                             else if(i==(dimensao-1)){
                                                     f[i]=(3-2*ponto[i])*ponto[i] - ponto[i-1] + 1;
                                                     }
                             else{
                                  f[i]=(3-2*ponto[i])*ponto[i] - ponto[i-1] - 2*ponto[i+1] + 1;
                                  }
                                   }

       //Jacobiana
```

```
        //preencher a matriz com zeros
       for(j=0;j<numero_funcoes;j++){
            for(i=0;i<dimensao;i++){
                                  J[i][j]=0;
                                  }
                                  }
       //Constroi a Jacobiana
       for(i=0;i<dimensao;i++){
                     J[i][i]=3-4*ponto[i];
                     J[i+1][i]=-1;
                     J[i][i+1]=-2;

                                 } //fecha for Jacobiana

       mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);

      //Calcula Valor da Funcao f(x)
      *f_x = 0;
      for(i=0;i<dimensao;i++){
         *f_x = *f_x + pow(f[i],2);
         }

} // fim rotina funcao broyden tridiagonal singular


double funcao_teste_desconhecida(int dimensao, int numero_funcoes, double ponto[TAM], double gradiente[TAM], double *f_x){
       int i,j;
       double f[TAM], J[TAM][TAM];

       //Eliminar possiveis lixos
       for(j=0;j<dimensao;j++){
           for(i=0;i<numero_funcoes;i++){
                                       J[i][j]=0;
                                       }
                               }
        for(i=0;i<numero_funcoes;i++){
                                    f[i]=0;
                                    }
```

```
      //f‘s
      f[0]= pow(ponto[0],2) + ponto[1] - 11;
      f[1]= ponto[0] + pow(ponto[1],2) - 7;

      //Jacobiana
      J[0][0]=2*ponto[0];             J[0][1]= 1;
      J[1][0]=1;                      J[1][1]= 2*ponto[1];

      mult_vetor_matriz(dimensao,numero_funcoes,f,J,gradiente);

      //Calcula Valor da Funcao f(x)
      *f_x = 0;
      *f_x = pow(f[0],2) + pow(f[1],2);

} // fim rotina funcao teste desconhecida



double funcao_six_hump_camel_modificada(double ponto[2], double gradiente[2]){
       int i,j;


       gradiente[0] = (-4.2*ponto[0] - ((4*pow(ponto[0],3))/3))*pow(ponto[0],2) - 2*ponto[0]*(-4 + 2.1*pow(ponto[0],2)
       - (pow(ponto[0],4)/3) ) + ponto[1];
       gradiente[1] = ponto[0] - (-8*pow(ponto[1],3) + 2*ponto[1]*(4 - 4*pow(ponto[1],2)));

} // fim rotina funcao six-hump_camel_modificada




double gradientes_conjugados(){

       FILE* arquivo;
       int i, j, k=0, z=0, opcao, sai=0, cont=0, dim, numero_funcoes, escolha_beta,  t=1;
       int arq_iteracao;
       double epsilon=0.00001, norma=0, resul_norma=0, f_x=0, cte_alpha=1;
```

```
      double g[TAM], g_anterior[TAM], gd, dir[TAM], dir_anterior[TAM];
      double f_LE[TAM], f_LD[TAM] ,f[TAM], sub_g_gant[TAM];
      double alpha=1, lambda=0.0001, beta=0, beta_FR, beta_PR, beta_x, LE=0, LD=0;
      double LD_aux=0, numerador=0, denominador=0, numerador_FR, denominador_FR;
      double numerador_PR, denominador_PR;
      double point[TAM], point_anterior[TAM];
      clock_t tInicio, tFim;
      double tDecorrido;
      double arq_norma=10000000, arq_tempo;
      double arq_ponto[TAM];


 printf("*********************************************************************************\n");
 printf("*********************** Programa Minimizacao de Funcao **********************\n");
 printf("*********************************************************************************\n");
 printf("************************* Metodo Gradientes Conjugados ************************\n");
 printf("*********************************************************************************\n");

while (!sai){

    printf("\n\t1. Quadratica -> x^2 + y^2 + ...\n");
    printf("\t2. Rosenbrock\n");
    printf("\t3. Freudenstein-Roth\n");
    printf("\t4. Powell Badly Scaled\n");
    printf("\t5. Brown Badly Scaled\n");
    printf("\t6. Rosenbrock Extended\n");
    printf("\t7. Powell Extended Singular\n");
    printf("\t8. Broyden Tridiagonal\n");
    printf("\t9. Funcao Teste Autor Desconhecido\n");
    printf("\t10. Funcao Six-hump Camel Modificada\n");
    printf("\t11. Sai do programa\n");
    printf("\n\tDigite sua opcao: ");
    scanf("%d", &opcao);

    //*****************  Opcao de Funcoes  *******************
    switch (opcao){
       case 1: printf("\n*********************************************************************************\n");

    //DIMENSAO DO PROBLEMA
```

```
      printf("\t******Entre com a dimensao******");
      printf("\n\n\tDimensao: ");
      scanf("%d",&dim);

      numero_funcoes=dim;

      printf("\n\t******Entre com o ponto inicial******");
      //**********************Dados iniciais*********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\n\tDigite o valor de x[%d]: ",i+1);
      scanf("%lf",&point[i]);
                        }
k=0;
norma=0;

//Calcula o gradiente com ponto inicial
funcao_quadratica(dim,numero_funcoes,point,g,&f_x);

//direcao inicial
      for(i=0;i<dim;i++){
                         dir[i]=-g[i];
                         }
      for(i=0;i<dim;i++){
                         printf("\n\tdirecao[%d]=%1.6f \n",i+1,dir[i]);
                         }
//************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);



   while(norma>=epsilon){
                         printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está
       //Guarda o gradiente  e direcao
       for(i=0;i<dim;i++){
```

```
                    g_anterior[i] = g[i];
                    dir_anterior[i]=dir[i];
                    }

  //calcula alpha com busca linear backtracking *********************************
  alpha=1;  LE=0; LD=0;

  for(i=0;i<dim;i++){
                    LE = LE + pow(point[i]+alpha*dir[i],2);
                    LD = LD + pow(point[i],2) + lambda*alpha*(g[i]*dir[i]);
                    }
  //printf("\n\tLE=%lf",LE);
  //printf("\n\tLD=%lf",LD);
 cont=0;
//Armijo
 while(LE >= LD){
        LE=0; LD=0;
    alpha=alpha/2;
        for(j=0;j<dim;j++){
                    LE = LE + pow(point[j]+alpha*dir[j],2);
                    LD = LD + pow(point[j],2) + lambda*alpha*(g[j]*dir[j]);
                    }
     }
     //printf("\ncontador=%d\n",cont);
     printf("\n\talpha=%lf",alpha);


  //calcula novo ponto
  for(i=0;i<dim;i++){
                    point[i] = point[i] + alpha*dir[i];
                    printf("\n\n\tx[%d]=%lf",i+1,point[i]);
                    }

  //calcula novo gradiente(k+1)
  funcao_quadratica(dim,numero_funcoes,point,g,&f_x);

 //calcula Beta
 calcula_beta(dim,escolha_beta,t,g,g_anterior,&beta);
```

```
      //calcula nova direcao (k+1)
      for(j=0;j<dim;j++){
                          dir[j]=-g[j]+beta*dir_anterior[j];
                          }

//Incrementa k e t
k=k+1;
t=t+1;

  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);

   } // fim while


     //Imprimi Solucao
     printf("\n\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                         printf("\t\tx[%d]=%lf",j+1,point[j]);
                         }

     //Valor da Funcao f(x)
     funcao_quadratica(dim,numero_funcoes,point,g,&f_x); //Recalcula f(x) no minimizador
     printf("\n\n\tf(x)=%lf",f_x);

     printf("\n\n*********************************************************************************\n\n");
                  break;

         case 2: printf("\n*********************************************************************************\n");
  //********* Cria um arquivo txt com as solucoes *****************
arquivo = fopen("Rosenbrock_dimensao_2.txt", "w+t");
if(arquivo!=NULL)
{
      //Rosenbrock com 2 dimensoes e 2 f‘s
      dim=2;
      numero_funcoes=2;
```

```
      printf("\t******Entre com o ponto inicial******\n\n");
      //**********************Dados iniciais*********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\tDigite o valor de x[%d]: ",i+1); ;
      scanf("%lf",&point[i]);
                        }
      printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
      scanf("%d",&escolha_beta);
k=0;
norma=0;

tInicio = clock();

//Calcula o gradiente com ponto inicial
funcao_rosenbrock(dim,numero_funcoes,point,g,&f_x);

//direcao inicial
      for(i=0;i<dim;i++){
                         dir[i]=-g[i];
                         }
      for(i=0;i<dim;i++){
                         printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                         }
//************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);


   while((norma>=epsilon)){  //   k<100  &&(k<50000)
             printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está

       //Guarda o gradiente  e direcao
       for(i=0;i<dim;i++){
                          g_anterior[i]=0; //Eliminar lixo
```

```
                        g_anterior[i]=g[i];
                        dir_anterior[i]=0; //Eliminar lixo
                        dir_anterior[i]=dir[i];
                        }

     //calcula alpha com busca linear backtracking ***************************

     alpha=1; LE=0; LD=0;
     LE=pow(10*((point[1]+alpha*dir[1])-pow(point[0]+alpha*dir[0],2)),2) + pow(1-point[0]+alpha*dir[0],2);
     //printf("\n\tLE=%lf",LE);
     mult_vetores(dim,g,dir,&gd);
     LD=pow(10*(point[1]-pow(point[0],2)),2)+pow(1-point[0],2)+ lambda*alpha*(gd);
     //printf("\n\tLD=%lf",LD);
cont=0;
   //Armijo
    while((LE > LD)&&(cont<=12)  ){ //  || (alpha >= 0.01)
       alpha=alpha/2;
           LE=pow(10*((point[1]+alpha*dir[1])-pow(point[0]+alpha*dir[0],2)),2) + pow(1-point[0]+alpha*dir[0],2);
           //printf("\n\tLE=%lf",LE);
           mult_vetores(dim,g,dir,&gd);
           LD=pow(10*(point[1]-pow(point[0],2)),2)+pow(1-point[0],2)+ lambda*alpha*(gd);
           //printf("\n\tLD=%lf",LD);
       cont=cont+1;
        }
        //printf("\n\tLE=%lf",LE); printf("\nLD=%lf",LD); printf("\ncontador=%d\n",cont); printf("\nalpha=%lf\n",alpha);

    //*********************************************************************


      //calcula novo ponto
     for(i=0;i<dim;i++){
                        point[i]=point[i]+alpha*dir[i];
                        printf("\n\n\tx[%d]= %lf",i+1,point[i]);
                        }

     //calcula novo gradiente(k+1)
     funcao_rosenbrock(dim,numero_funcoes,point,g,&f_x);
```

```
      //calcula Beta
      calcula_beta(dim,escolha_beta,t,g,g_anterior,&beta);


      //calcula nova direcao (k+1)
      for(j=0;j<dim;j++){
                          dir[j]=-g[j]+beta*dir_anterior[j];
                          }

//Incrementa k e t
k=k+1;
t=t+1;

  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);

   //Arquivando uma possivel solucao num arquivo txt
   if(norma<=arq_norma){
                        arq_norma=norma;
                        for(j=0;j<dim;j++){
                                           arq_ponto[j] = point[j];
                                           }
                        arq_iteracao = k;
                        arq_tempo = ((clock() - tInicio)/CLOCKS_PER_SEC);
                        }
} // fim while

tFim = clock();


/*calcula aproximadamente o tempo em milisegundos gasto entre as duas chamadas de clock()*/
tDecorrido = ((tFim - tInicio) / (CLOCKS_PER_SEC));

           //Imprime tempo gasto para o cálculo da solucao
            printf("\n\nTempo gasto: %lf seg\n",tDecorrido);

     //Imprimi Solucao
```

```
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                         printf("\t\tx[%d]=%lf",j+1,point[j]);
                         }

   //Imprimi as informacoes no arquivo texto
    fprintf(arquivo,"\niteracao=%d\n",arq_iteracao);
    fprintf(arquivo,"\nnorma=%lf\n",arq_norma);
    fprintf(arquivo,"\ntempo=%lf\n",arq_tempo);

    for(j=0;j<dim;j++){
                        fprintf(arquivo,"\nx[%d]=%lf\t",j+1,arq_ponto[j]);
                         }


     //Valor da Funcao f(x)
     funcao_rosenbrock(dim,numero_funcoes,point,g,&f_x); //Recalcula f(x) no minimizador
     printf("\n\n\tf(x)=%2.13lf",f_x);

     printf("\n\n*********************************************************************************\n\n");
}fclose(arquivo);
                    break;

         case 3: printf("\n*********************************************************************************\n");

  //********* Cria um arquivo txt com as solucoes *****************
arquivo = fopen("Freudenstein_Roth.txt", "w+t");
if(arquivo!=NULL)
{

      //Freudenstein-Roth com 2 dimensoes e 2 f's
      dim=2;
      numero_funcoes=dim;

      printf("\t******Entre com o ponto inicial******\n\n");
      //**********************Dados iniciais*********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\tDigite o valor de x[%d]: ",i+1); ;
```

```
      scanf("%lf",&point[i]);
                        }
      printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
      scanf("%d",&escolha_beta);
k=0;
t=1;
norma=0;

tInicio = clock();

//Calcula o gradiente com ponto inicial
funcao_freudenstein_roth(dim,numero_funcoes,point,g,&f_x);

//direcao inicial
      for(i=0;i<dim;i++){
                        dir[i]=-g[i];
                        }
      for(i=0;i<dim;i++){
                        printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                        }
//************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);

   while(norma>=epsilon){
                        printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está

       //Guarda o gradiente  e direcao
       for(i=0;i<dim;i++){
                         g_anterior[i]=0; //Eliminar lixo
                         g_anterior[i]=g[i];
                         dir_anterior[i]=0; //Eliminar lixo
                         dir_anterior[i]=dir[i];
                         }

      //calcula alpha com busca linear backtracking ***************************
```

```
      alpha=1; LE=0; LD=0;
      LE=pow(-13+point[0]+alpha*dir[0]+ 5*pow(point[1]+alpha*dir[1],2)-pow(point[1]+alpha*dir[1],3)
      -2*(point[1]+alpha*dir[1]),2)+pow(-29+point[0]+alpha*dir[0]+pow(point[1]+alpha*dir[1],3)+
      +pow(point[1]+alpha*dir[1],2)-14*(point[1]+alpha*dir[1]),2);
      //printf("\nLE=%lf",LE);
      mult_vetores(dim,g,dir,&gd);
      LD=pow(-13+point[0]+5*pow(point[1],2)-pow(point[1],3)-2*point[1],2)+pow(-29+point[0]+pow(point[1],3)+
      +pow(point[1],2)-14*point[1],2)+lambda*alpha*(gd);
      //printf("\nLD=%lf",LD);
cont=0;
   //Armijo
    while((LE > LD)){ // || (alpha >= 0.01)  &&(cont<=12)
       alpha=alpha/2;
       LE=pow(-13+point[0]+alpha*dir[0]+ 5*pow(point[1]+alpha*dir[1],2)-pow(point[1]+alpha*dir[1],3)-
       -2*(point[1]+alpha*dir[1]),2)+
       +pow(point[1]+alpha*dir[1],2)-14*(point[1]+alpha*dir[1]),2);
       //printf("\nLE=%lf",LE);
       mult_vetores(dim,g,dir,&gd);
       LD=pow(-13+point[0]+5*pow(point[1],2)-pow(point[1],3)-2*point[1],2)+pow(-29+point[0]+pow(point[1],3)+
       +pow(point[1],2)-14*point[1],2)+lambda*alpha*(gd);
       //printf("\nLD=%lf",LD);
       cont=cont+1;
        }
        //printf("\nLE=%lf",LE);
        //printf("\nLD=%lf",LD);
        //printf("\ncontador=%d\n",cont);
        //printf("\nalpha=%lf\n",alpha);

    //*********************************************************************


      //calcula novo ponto
     for(i=0;i<dim;i++){
                        point[i]=point[i]+alpha*dir[i];
                        printf("\n\n\tx[%d]= %lf",i+1,point[i]);
                        }

     //calcula novo gradiente(k+1)
     funcao_freudenstein_roth(dim,numero_funcoes,point,g,&f_x);
```

```
       //calcula Beta
       calcula_beta(dim,escolha_beta,t,g,g_anterior,&beta);

      //calcula nova direcao (k+1)
      for(j=0;j<dim;j++){
                          dir[j]=-g[j]+beta*dir_anterior[j];
                          }

//Incrementa k e t
k=k+1;
t=t+1;


  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);


   //Imprimi as informacoes no arquivo texto
   fprintf(arquivo,"\niteracao=%d\n",k);
      for(j=0;j<dim;j++){
                         fprintf(arquivo,"\nx[%d]=%1.10lf\t",j+1,point[j]);
                          }



   } // fim while

tFim = clock();


/*calcula aproximadamente o tempo em milisegundos gasto entre as duas chamadas de clock()*/
tDecorrido = ((tFim - tInicio) / (CLOCKS_PER_SEC));


           //Imprime tempo gasto para o cálculo da solucao
```

```
            printf("\n\nTempo gasto: %lf seg\n",tDecorrido);


     //Imprimi Solucao
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                        printf("\t\tx[%d]=%lf",j+1,point[j]);
                        }

     //Valor da Funcao f(x)
     funcao_freudenstein_roth(dim,numero_funcoes,point,g,&f_x);//Recalcula f(x) no minimizador
     printf("\n\n\tf(x)=%2.13lf",f_x);

printf("\n\n*********************************************************************************\n\n");
}fclose(arquivo);

                   break;

         case 4: printf("\n*********************************************************************************\n\n");

      //Powell com 2 dimensoes e 2 f's
      dim=2;
      numero_funcoes=dim;

      printf("\t******Entre com o ponto inicial******\n\n");
      //**********************Dados iniciais*********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\tDigite o valor de x[%d]: ",i+1); ;
      scanf("%lf",&point[i]);
                        }
      printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
      scanf("%d",&escolha_beta);
k=0;
norma=0;

//Calcula o gradiente com ponto inicial
funcao_powell(dim,numero_funcoes,point,g,&f_x);
```

```
//direcao inicial
      for(i=0;i<dim;i++){
                          dir[i]=-g[i];
                          }
      for(i=0;i<dim;i++){
                          printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                          }
//************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);



   while(norma>=epsilon){
            printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está

       //Guarda o gradiente  e direcao
       for(i=0;i<dim;i++){
                           g_anterior[i]=0; //Eliminar lixo
                           g_anterior[i]=g[i];
                           dir_anterior[i]=0; //Eliminar lixo
                           dir_anterior[i]=dir[i];
                           }

      //calcula alpha com busca linear backtracking ****************************
       alpha=1; LE=0; LD=0;
       LE=pow(pow(10,4)*(point[0]+alpha*dir[0])*(point[1]+alpha*dir[1])-1,2) + pow(exp(-(point[0]+alpha*dir[0]))+
       +exp(-(point[1]+alpha*dir[1]))-1.0001,2);
       //printf("\nLE=%lf",LE);
       mult_vetores(dim,g,dir,&gd);
       LD=pow(pow(10,4)*point[0]*point[1]-1,2) + pow(exp(-point[0])+exp(-point[1])-1.0001,2) + lambda*alpha*(gd);
       //printf("\nLD=%lf",LD);
  cont=0;
     //Armijo
      while((LE > LD)&&(cont<=12)){ // || (alpha >= 0.01)
         alpha=alpha/2;
         LE=pow(pow(10,4)*(point[0]+alpha*dir[0])*(point[1]+alpha*dir[1])-1,2) + pow(exp(-(point[0]+alpha*dir[0]))+
```

```
   +exp(-(point[1]+alpha*dir[1]))-1.0001,2);
   //printf("\nLE=%lf",LE);
   mult_vetores(dim,g,dir,&gd);
   LD=pow(pow(10,4)*point[0]*point[1]-1,2) + pow(exp(-point[0])+exp(-point[1])-1.0001,2) + lambda*alpha*(gd);
   //printf("\nLD=%lf",LD);
   cont=cont+1;
    }
    //printf("\nLE=%lf",LE);
    //printf("\nLD=%lf",LD);
    //printf("\ncontador=%d\n",cont);
    printf("\nalpha=%lf\n",alpha);

//*********************************************************************

  //calcula novo ponto
 for(i=0;i<dim;i++){
                   point[i]=point[i]+alpha*dir[i];
                   printf("\n\n\tx[%d]= %lf",i+1,point[i]);
                   }

 //calcula novo gradiente(k+1)
 funcao_powell(dim,numero_funcoes,point,g,&f_x);

//calcula Beta

       if(escolha_beta==1){

      //Fletcher-Reeves
                        mult_vetores(dim,g,g,&numerador);
                        //printf("\nnumerador=%lf",numerador);
                        mult_vetores(dim,g_anterior,g_anterior,&denominador);
                        //printf("\ndenominador=%lf",denominador);
           beta=(numerador)/(denominador);
         //printf("\n\nbeta=%lf\n",beta);
                                       }
       else if(escolha_beta==2){

     //Polak-Ribiere
                   for(i=0;i<dim;i++){
```

```
                                        sub_g_gant[i]=0; //Eliminar possiveis lixos
                                        sub_g_gant[i]=g[i]-g_anterior[i];
                                        }
                        mult_vetores(dim,g,sub_g_gant,&numerador);
                        mult_vetores(dim,g_anterior,g_anterior,&denominador);

                beta=(numerador)/(denominador);
                                }
            else{

             //Dai-Yuan
                        for(i=0;i<dim;i++){
                                        sub_g_gant[i]=0; //Eliminar possiveis lixos
                                        sub_g_gant[i]=g[i]-g_anterior[i];
                                        }
                        mult_vetores(dim,g,g,&numerador);
                        mult_vetores(dim,dir,sub_g_gant,&denominador);

                beta=(numerador)/(denominador);
                }

      //calcula nova direcao (k+1)
      for(j=0;j<dim;j++){
                        dir[j]=-g[j]+beta*dir_anterior[j];
                        }

//Incrementa k
k=k+1;

  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);

   } // fim while


     //Imprimi Solucao
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
```

```
     for(j=0;j<dim;j++){
                         printf("\t\tx[%d]=%lf",j+1,point[j]);
                         }

printf("\n\n*********************************************************************************\n\n");
                    break;

          case 5: printf("\n*********************************************************************************\n\n");

       //Brown com 2 dimensoes e 3 f‘s
       dim=2;
       numero_funcoes=3;

       printf("\t******Entre com o ponto inicial******\n\n");
       //**********************Dados iniciais*********************
       // ponto inicial
       for(i=0;i<dim;i++){
       printf("\tDigite o valor de x[%d]: ",i+1); ;
       scanf("%lf",&point[i]);
                         }
        printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
       scanf("%d",&escolha_beta);



k=0;
norma=0;

 //Calcula o gradiente com ponto inicial
 funcao_brown(dim,numero_funcoes,point,g);

//direcao inicial
       for(i=0;i<dim;i++){
                          dir[i]=-g[i];
                          }
       for(i=0;i<dim;i++){
                          //printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                          printf("\n\tgradiente[%d]=%lf \n",i+1,g[i]);
                          }
```

```
//*********************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);

   while(norma>=epsilon){
                        printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está

       //Guarda o gradiente , direcao e ponto
       for(i=0;i<dim;i++){
                          g_anterior[i]=0; //Eliminar possiveis lixo
                          g_anterior[i]=g[i];
                          //printf("\n\tg_anterior[%d]=%lf",i+1,g_anterior[i]);
                          dir_anterior[i]=0; //Eliminar possiveis lixo
                          dir_anterior[i]=dir[i];
                          printf("\n\tdir_anterior[%d]=%lf",i+1,dir_anterior[i]);
                          point_anterior[i]=0; //Eliminar possiveis lixo
                          point_anterior[i]=point[i];
                          }

      //calcula alpha com busca linear backtracking ***************************
       alpha=1; LE=0; LD=0;
       LE=pow((point[0]+alpha*dir[0])-pow(10,6),2)+pow((point[1]+alpha*dir[1])-2*pow(10,-6),2)+pow((point[0]+
       +alpha*dir[0])*(point[1]+alpha*dir[1])-2,2);
       //printf("\nLE=%lf",LE);
       mult_vetores(dim,g,dir,&gd);
       LD=pow(point[0]-pow(10,6),2)+pow(point[1]-2*pow(10,-6),2)+pow(point[0]*point[1]-2,2) + lambda*alpha*(gd);
       //printf("\nLD=%lf",LD);
       cont=0;
      //Armijo
       while((LE>LD)&&(cont<=12)){ // || (alpha >= 0.01)
            LE=0; LD=0; alpha=alpha/2;
          LE=pow((point[0]+alpha*dir[0])-pow(10,6),2)+pow((point[1]+alpha*dir[1])-2*pow(10,-6),2)+
          +pow((point[0]+alpha*dir[0])*(point[1]+alpha*dir[1])-2,2);
          //printf("\nLE=%lf",LE);
          mult_vetores(dim,g,dir,&gd);
          LD=pow(point[0]-pow(10,6),2)+pow(point[1]-2*pow(10,-6),2)+pow(point[0]*point[1]-2,2) + lambda*alpha*(gd);
```

```
   //printf("\nLD=%lf",LD);
   cont=cont+1;
    }
    //printf("\nLE=%lf",LE);
    //printf("\nLD=%lf",LD);
    //printf("\ncontador=%d\n",cont);
    printf("\n\n\talpha=%lf\n",alpha);

//*********************************************************************

   //calcula novo gradiente(k+1)
 funcao_brown(dim,numero_funcoes,point,g);

       //Imprime o gradiente
       for(i=0;i<dim;i++){
                   printf("\n\n\tgradiente[%d]=%lf",i+1,g[i]);
                   }

  //calcula novo ponto
 for(i=0;i<dim;i++){
                    point[i]=0; //Elimina possiveis lixo
                    point[i]=point_anterior[i]+alpha*dir[i];
                    printf("\n\tx[%d]= %7.6f",i+1,point[i]);
                    }


 //calcula novo gradiente(k+1)
 funcao_brown(dim,numero_funcoes,point,g);

       //Imprime o gradiente
      // for(i=0;i<dim;i++){
                  // printf("\n\n\tgradiente[%d]=%lf",i+1,g[i]);
                  // }


 //calcula Beta

       if(escolha_beta==1){
```

```
      //Fletcher-Reeves
                        mult_vetores(dim,g,g,&numerador);
                        //printf("\nnumerador=%lf",numerador);
                        mult_vetores(dim,g_anterior,g_anterior,&denominador);
                        //printf("\ndenominador=%lf",denominador);
            beta=(numerador)/(denominador);
           //printf("\n\nbeta=%lf\n",beta);
                                        }
       else if(escolha_beta==2){

     //Polak-Ribiere
                    for(i=0;i<dim;i++){
                                        sub_g_gant[i]=0; //Eliminar possiveis lixos
                                        sub_g_gant[i]=g[i]-g_anterior[i];
                                        }
                    mult_vetores(dim,g,sub_g_gant,&numerador);
                    mult_vetores(dim,g_anterior,g_anterior,&denominador);

            beta=(numerador)/(denominador);
                                }
       else{

        //Dai-Yuan
                    for(i=0;i<dim;i++){
                                        sub_g_gant[i]=0; //Eliminar possiveis lixos
                                        sub_g_gant[i]=g[i]-g_anterior[i];
                                        }
                    mult_vetores(dim,g,g,&numerador);
                    mult_vetores(dim,dir,sub_g_gant,&denominador);

            beta=(numerador)/(denominador);
            }


//calcula nova direcao (k+1) ******************************************
for(j=0;j<dim;j++){
                    dir[j]=0; //Eliminar possiveis lixo
                    dir[j]=-g[j]+beta*dir_anterior[j];
                    }
```

```
            for(j=0;j<dim;j++){
                        printf("\n\tdirecao[%d]=%lf \n",j+1,dir[j]);
                        }


//Incrementa k
k=k+1;

       //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
       resul_norma=0; //Eliminar possiveis lixo
       mult_vetores(dim,g,g,&resul_norma);
       norma=0; //Eliminar possiveis lixo
       norma=sqrt(resul_norma);
       printf("\n\n\tnorma=%lf",norma);



   } // fim while


     //Imprimi Solucao
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                        printf("\t\tx[%d]=%lf",j+1,point[j]);
                        }

printf("\n\n*********************************************************************************\n\n");
                  break;

        case 6:  printf("\n*********************************************************************************\n");

  //********* Cria um arquivo txt com as solucoes *****************
arquivo = fopen("Rosenbrock_Extended.txt", "w+t");
if(arquivo!=NULL)
{
      //DIMENSAO DO PROBLEMA
      printf("\t******Entre com a dimensao******");
      printf("\n\n\tDimensao: ");
```

```
      scanf("%d",&dim);

      numero_funcoes=dim;

      printf("\n\t******Entre com o ponto inicial******\n\n");
      //**********************Dados iniciais*********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\tDigite o valor de x[%d]: ",i+1); ;
      scanf("%lf",&point[i]);
                        }
      printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
      scanf("%d",&escolha_beta);
k=0;
norma=0;

tInicio = clock();

//Calcula o gradiente com ponto inicial
funcao_rosenbrock_extended(dim,numero_funcoes,point,g,&f_x);

//direcao inicial
      for(i=0;i<dim;i++){
                        dir[i]=-g[i];
                        }
      for(i=0;i<dim;i++){
                        printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                        }
//************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);


   while(norma>=epsilon){
             printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está
```

```
//Guarda o gradiente  e direcao
for(i=0;i<dim;i++){
                  g_anterior[i]=0; //Eliminar lixo
                  g_anterior[i]=g[i];
                  dir_anterior[i]=0; //Eliminar lixo
                  dir_anterior[i]=dir[i];
                  }

//calcula alpha com busca linear backtracking ****************************

//Eliminar possiveis lixo
for(j=0;j<dim;j++){
                  f_LE[j]=0;
                  f_LD[j]=0;
                  }


alpha=1; LE=0; LD=0; LD_aux=0;                                          //+alpha*dir[]
   //Lado Esquerdo Armijo
      for(i=0;i<(dim/2);i++){
                     f_LE[2*i]=10*((point[2*i+1]+alpha*dir[2*i+1]) - pow((point[2*i]+alpha*dir[2*i]),2));
                     f_LE[2*i+1]=1-(point[2*i]+alpha*dir[2*i]);
                           }

      for(j=0;j<dim;j++){
                       LE = LE + pow(f_LE[j],2);
                       }

//printf("\n\tLE=%lf",LE);

  //Lado Direito

      for(i=0;i<(dim/2);i++){
                      f_LD[2*i]=10*(point[2*i+1] - pow(point[2*i],2));
                      f_LD[2*i+1]=1-point[2*i];
                          }
      for(j=0;j<dim;j++){
                       LD_aux = LD_aux + pow(f_LD[j],2);
                       }
```

```
                    mult_vetores(dim,g,dir,&gd);
                    LD = LD_aux + lambda*alpha*(gd);

     //printf("\n\tLD=%lf",LD);
  cont=0;
    //Armijo
     while((LE > LD)&&(cont<=12)){
        alpha=alpha/2;
          LE;
              LD = LD + lambda*alpha*(gd);
        cont=cont+1;
         }
         //printf("\n\tLE=%lf",LE); printf("\nLD=%lf",LD); printf("\ncontador=%d\n",cont); printf("\nalpha=%lf\n",alpha);

     //**********************************************************************



       //calcula novo ponto
      for(i=0;i<dim;i++){
                         point[i]=point[i]+alpha*dir[i];
                         printf("\n\n\tx[%d]= %lf",i+1,point[i]);
                         }

      //calcula novo gradiente(k+1)
      funcao_rosenbrock_extended(dim,numero_funcoes,point,g,&f_x);



      //calcula Beta
      calcula_beta(dim,escolha_beta,t,g,g_anterior,&beta);



     //calcula nova direcao (k+1)
     for(j=0;j<dim;j++){
                        dir[j]=-g[j]+beta*dir_anterior[j];
                        }

//Incrementa k e t
```

```
k=k+1;
t=t+1;

  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);

   //Arquivando uma possivel solucao num arquivo txt
   if(norma<=arq_norma){
                        arq_norma=norma;
                        for(j=0;j<dim;j++){
                                           arq_ponto[j] = point[j];
                                           }
                        arq_iteracao = k;
                        arq_tempo = ((clock() - tInicio)/CLOCKS_PER_SEC);
                        }

} // fim while

tFim = clock();


/*calcula aproximadamente o tempo em milisegundos gasto entre as duas chamadas de clock()*/
tDecorrido = ((tFim - tInicio) / (CLOCKS_PER_SEC));

           //Imprime tempo gasto para o cálculo da solucao
            printf("\n\nTempo gasto: %lf seg\n",tDecorrido);

     //Imprimi Solucao
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                        printf("\t\tx[%d]=%lf",j+1,point[j]);
                        }


   //Imprimi as informacoes no arquivo texto
    fprintf(arquivo,"\niteracao=%d\n",arq_iteracao);
    fprintf(arquivo,"\nnorma=%lf\n",arq_norma);
```

```
    fprintf(arquivo,"\ntempo=%lf\n",arq_tempo);
   for(j=0;j<dim;j++){
                       fprintf(arquivo,"\nx[%d]=%lf\t",j+1,arq_ponto[j]);
                        }

     //Valor da Funcao f(x)
     funcao_rosenbrock_extended(dim,numero_funcoes,point,g,&f_x); //Recalcula f(x) no minimizador
     printf("\n\n\tf(x)=%2.13lf",f_x);

     printf("\n\n*********************************************************************************\n\n");
}fclose(arquivo);
                  break;

        case 7:  printf("\n*********************************************************************************\n");

  //********* Cria um arquivo txt com as solucoes *****************
arquivo = fopen("Powell_Extended_Singular.txt", "w+t");
if(arquivo!=NULL)
{

      //DIMENSAO DO PROBLEMA
      printf("\t******Entre com a dimensao******");
      printf("\n\n\tDimensao: ");
      scanf("%d",&dim);

      numero_funcoes=dim;

      printf("\n\t******Entre com o ponto inicial******\n\n");
      //**********************Dados iniciais**********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\tDigite o valor de x[%d]: ",i+1); ;
      scanf("%lf",&point[i]);
                       }
      printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
      scanf("%d",&escolha_beta);
k=0;
norma=0;
```

```
tInicio = clock();

//Calcula o gradiente com ponto inicial
funcao_powell_extended(dim,numero_funcoes,point,g,&f_x);

//direcao inicial
      for(i=0;i<dim;i++){
                          dir[i]=-g[i];
                          }
      for(i=0;i<dim;i++){
                          printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                          }
//*************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);


   while((norma>=epsilon)&&(k<=50000)){  //||(z<=3500000) (norma>=epsilon)
             printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está

       //Guarda o gradiente  e direcao
       for(i=0;i<dim;i++){
                           g_anterior[i]=0; //Eliminar lixo
                           g_anterior[i]=g[i];
                           dir_anterior[i]=0; //Eliminar lixo
                           dir_anterior[i]=dir[i];
                           }

       //calcula alpha com busca linear backtracking ****************************

       //Eliminar possiveis lixo
       for(j=0;j<dim;j++){
                           f_LE[j]=0;
                           f_LD[j]=0;
                           }
```

```
     alpha=1; LE=0; LD=0; LD_aux=0;                                          //+alpha*dir[]
        //Lado Esquerdo Armijo
           for(i=0;i<(dim/4);i++){
                              f_LE[4*i]= (point[4*i]+alpha*dir[4*i]) + 10*(point[4*i+1]+alpha*dir[4*i+1]) ;
                              f_LE[4*i+1]=sqrt(5)*((point[4*i+2]+alpha*dir[4*i+2]) - (point[4*i+3]+alpha*dir[4*i+3])) ;
                              f_LE[4*i+2]=pow((point[4*i+1]+alpha*dir[4*i+1]) - 2*(point[4*i+2]+alpha*dir[4*i+2]),2);
                              f_LE[4*i+3]=sqrt(10)*pow(((point[4*i]+alpha*dir[4*i]) - (point[4*i+3]+alpha*dir[4*i+3])),2);
                                  }

           for(j=0;j<dim;j++){
                               LE = LE + pow(f_LE[j],2);
                               }

     //printf("\n\tLE=%lf",LE);

       //Lado Direito

           for(i=0;i<(dim/4);i++){
                              f_LD[4*i]= point[4*i] + 10*point[4*i+1] ;
                              f_LD[4*i+1]=sqrt(5)*(point[4*i+2] - point[4*i+3]) ;
                              f_LD[4*i+2]=pow(point[4*i+1] - 2*point[4*i+2],2);
                              f_LD[4*i+3]=sqrt(10)*pow((point[4*i] - point[4*i+3]),2);
                                  }
           for(j=0;j<dim;j++){
                               LD_aux = LD_aux + pow(f_LD[j],2);
                               }

                    mult_vetores(dim,g,dir,&gd);
                    LD = LD_aux + lambda*alpha*(gd);

     //printf("\n\tLD=%lf",LD);
cont=0;
   //Armijo
    while((LE > LD)&&(cont<=12)){
       alpha=alpha/2;
         LE;
             LD = LD + lambda*alpha*(gd);
       cont=cont+1;
```

```
          }
          //printf("\n\tLE=%lf",LE); printf("\nLD=%lf",LD); printf("\ncontador=%d\n",cont); printf("\nalpha=%lf\n",alpha);

      //*********************************************************************



        //calcula novo ponto
       for(i=0;i<dim;i++){
                          point[i]=point[i]+alpha*dir[i];
                          printf("\n\n\tx[%d]= %lf",i+1,point[i]);
                          }

       //calcula novo gradiente(k+1)
       funcao_powell_extended(dim,numero_funcoes,point,g,&f_x);



       //calcula Beta
       calcula_beta(dim,escolha_beta,t,g,g_anterior,&beta);

      //calcula nova direcao (k+1)
      for(j=0;j<dim;j++){
                         dir[j]=-g[j]+beta*dir_anterior[j];
                         }

//Incrementa k e t
k=k+1;
t=t+1;

  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);

   //Arquivando uma possivel solucao num arquivo txt
   //if(norma<=arq_norma){
                        arq_norma=norma;
                        for(j=0;j<dim;j++){
                                           arq_ponto[j] = point[j];
```

```
                                        }
                        arq_iteracao = k;
                        arq_tempo = ((clock() - tInicio)/CLOCKS_PER_SEC);
                       // }

} // fim while

tFim = clock();


/*calcula aproximadamente o tempo em milisegundos gasto entre as duas chamadas de clock()*/
tDecorrido = ((tFim - tInicio) / (CLOCKS_PER_SEC));

           //Imprime tempo gasto para o cálculo da solucao
            printf("\n\nTempo gasto: %lf seg\n",tDecorrido);

     //Imprimi Solucao
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                        printf("\t\tx[%d]=%lf",j+1,point[j]);
                        }


   //Imprimi as informacoes no arquivo texto
    fprintf(arquivo,"\niteracao=%d\n",arq_iteracao);
    fprintf(arquivo,"\nnorma=%lf\n",arq_norma);
    fprintf(arquivo,"\ntempo=%lf\n",arq_tempo);
   for(j=0;j<dim;j++){
                       fprintf(arquivo,"\nx[%d]=%lf\t",j+1,arq_ponto[j]);
                        }

     //Valor da Funcao f(x)
     funcao_powell_extended(dim,numero_funcoes,point,g,&f_x);//Recalcula f(x) no minimizador
     printf("\n\n\tf(x)=%2.13lf",f_x);

     printf("\n\n*******************************************************************************\n\n");
}fclose(arquivo);
                   break;
```

```
        case 8:  printf("\n*************************************************************************************\n");

  //********* Cria um arquivo txt com as solucoes *****************
arquivo = fopen("Broyden_Extended.txt", "w+t");
if(arquivo!=NULL)
{

      //DIMENSAO DO PROBLEMA
      printf("\t******Entre com a dimensao******");
      printf("\n\n\tDimensao: ");
      scanf("%d",&dim);

      numero_funcoes=dim;



      //if(dim==36){
      //Pela dimensao ser muito grande fica cansativo entrar com 36 componentes do ponto inicial
      //entao é feito a entrada automatica dessas componentes com mesmos valores
      //for(j=0;j<dim;j++){
        //                point[j]=-1;
          //              }
            //      }
      //else{
      printf("\n\t******Entre com o ponto inicial******\n\n");
      //**********************Dados iniciais*********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\tDigite o valor de x[%d]: ",i+1); ;
      scanf("%lf",&point[i]);
                        }
        //   }
      printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
      scanf("%d",&escolha_beta);
k=0;
norma=0;

tInicio = clock();

//Calcula o gradiente com ponto inicial
```

```
funcao_broyden_tridiagonal(dim,numero_funcoes,point,g,&f_x);

//direcao inicial
      for(i=0;i<dim;i++){
                          dir[i]=-g[i];
                          }
      for(i=0;i<dim;i++){
                          printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                          }
//************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);



   while((norma>=epsilon)&&(k<=50000)){
             printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está

       //Guarda o gradiente  e direcao
       for(i=0;i<dim;i++){
                           g_anterior[i]=0; //Eliminar lixo
                           g_anterior[i]=g[i];
                           dir_anterior[i]=0; //Eliminar lixo
                           dir_anterior[i]=dir[i];
                           }

       //calcula alpha com busca linear backtracking ****************************

       //Eliminar possiveis lixo
       for(j=0;j<dim;j++){
                           f_LE[j]=0;
                           f_LD[j]=0;
                           }


       alpha=1; LE=0; LD=0; LD_aux=0;                                          //+alpha*dir[]
          //Lado Esquerdo Armijo
```

```
      for(i=0;i<dim;i++){
                      if(i==0){
                               f_LE[i]=(3-2*(point[i]+alpha*dir[i]))*(point[i]+alpha*dir[i])-
                                - 2*(point[i+1]+alpha*dir[i+1]) +1;
                               }
                      else if(i==(dim-1)){
                                                  f_LE[i]=(3-2*(point[i]+alpha*dir[i]))*(point[i]+alpha*dir[i])-
                                                   - (point[i-1]+alpha*dir[i-1]) + 1;
                                                  }
                      else{
                           f_LE[i]=(3-2*(point[i]+alpha*dir[i]))*(point[i]+alpha*dir[i])-
                            - (point[i-1]+alpha*dir[i-1]) - 2*(point[i+1]+alpha*dir[i+1]) + 1;
                           }
                            }

      for(j=0;j<dim;j++){
                         LE = LE + pow(f_LE[j],2);
                         }

//printf("\n\tLE=%lf",LE);

  //Lado Direito

      for(i=0;i<dim;i++){
                      if(i==0){
                               f_LD[i]=(3-2*point[i])*point[i] - 2*point[i+1] +1;
                               }
                      else if(i==(dim-1)){
                                                  f_LD[i]=(3-2*point[i])*point[i] - point[i-1] + 1;
                                                  }
                      else{
                           f_LD[i]=(3-2*point[i])*point[i] - point[i-1] - 2*point[i+1] + 1;
                           }
                            }
      for(j=0;j<dim;j++){
                         LD_aux = LD_aux + pow(f_LD[j],2);
                         }

               mult_vetores(dim,g,dir,&gd);
```

```
                    LD = LD_aux + lambda*alpha*(gd);

      //printf("\n\tLD=%lf",LD);
  cont=0;
     //Armijo
     while((LE > LD)&&(cont<=12)){
        alpha=alpha/2;
          LE;
              LD = LD + lambda*alpha*(gd);
        cont=cont+1;
         }
         //printf("\n\tLE=%lf",LE); printf("\nLD=%lf",LD); printf("\ncontador=%d\n",cont); printf("\nalpha=%lf\n",alpha);

     //**********************************************************************



       //calcula novo ponto
      for(i=0;i<dim;i++){
                         point[i]=point[i]+alpha*dir[i];
                         printf("\n\n\tx[%d]= %lf",i+1,point[i]);
                         }

      //calcula novo gradiente(k+1)
      funcao_broyden_tridiagonal(dim,numero_funcoes,point,g,&f_x);



      //calcula Beta
      calcula_beta(dim,escolha_beta,t,g,g_anterior,&beta);

     //calcula nova direcao (k+1)
     for(j=0;j<dim;j++){
                        dir[j]=-g[j]+beta*dir_anterior[j];
                        }

//Incrementa k e t
k=k+1;
t=t+1;
```

```
  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);


                        arq_norma=norma;
                        for(j=0;j<dim;j++){
                                          arq_ponto[j] = point[j];
                                          }
                        arq_iteracao = k;
                        arq_tempo = ((clock() - tInicio)/CLOCKS_PER_SEC);

} // fim while

tFim = clock();


/*calcula aproximadamente o tempo em milisegundos gasto entre as duas chamadas de clock()*/
tDecorrido = ((tFim - tInicio) / (CLOCKS_PER_SEC));

           //Imprime tempo gasto para o cálculo da solucao
            printf("\n\nTempo gasto: %lf seg\n",tDecorrido);

     //Imprimi Solucao
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                        printf("\t\tx[%d]=%lf",j+1,point[j]);
                        }

   //Imprimi as informacoes no arquivo texto
    fprintf(arquivo,"\niteracao=%d\n",arq_iteracao);
    fprintf(arquivo,"\nnorma=%lf\n",arq_norma);
    fprintf(arquivo,"\ntempo=%lf\n",arq_tempo);
   for(j=0;j<dim;j++){
                       fprintf(arquivo,"\nx[%d]=%lf\t",j+1,arq_ponto[j]);
                        }
```

```
     //Valor da Funcao f(x)
     funcao_broyden_tridiagonal(dim,numero_funcoes,point,g,&f_x);//Recalcula f(x) no minimizador
     printf("\n\n\tf(x)=%2.13lf",f_x);

     printf("\n\n*********************************************************************************\n\n");
}fclose(arquivo);

                  break;

         case 9: printf("\n*********************************************************************************\n");

  //********* Cria um arquivo txt com as solucoes *****************
arquivo = fopen("Funcao_Teste_8.txt", "w+t");
if(arquivo!=NULL)
{

      //Funcao Autor desconhecido com 2 dimensoes e 2 f's
      dim=2;
      numero_funcoes=dim;

      printf("\t******Entre com o ponto inicial******\n\n");
      //**********************Dados iniciais*********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\tDigite o valor de x[%d]: ",i+1); ;
      scanf("%lf",&point[i]);
                        }
      printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
      scanf("%d",&escolha_beta);
k=0;
norma=0;

tInicio = clock();

//Calcula o gradiente com ponto inicial
funcao_teste_desconhecida(dim,numero_funcoes,point,g,&f_x);

//direcao inicial
      for(i=0;i<dim;i++){
```

```
                        dir[i]=-g[i];
                        }
     for(i=0;i<dim;i++){
                        printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                        }
//************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);

   while(norma>=epsilon){
                        printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está

       //Guarda o gradiente  e direcao
       for(i=0;i<dim;i++){
                         g_anterior[i]=0; //Eliminar lixo
                         g_anterior[i]=g[i];
                         dir_anterior[i]=0; //Eliminar lixo
                         dir_anterior[i]=dir[i];
                         }

      //calcula alpha com busca linear backtracking ****************************
       alpha=1; LE=0; LD=0;
       LE = pow(pow(point[0]+alpha*dir[0],2) + (point[1]+alpha*dir[1]) - 11,2) + pow((point[0]+alpha*dir[0])+
        + pow(point[1]+alpha*dir[1],2) - 7,2);
       //printf("\nLE=%lf",LE);
       mult_vetores(dim,g,dir,&gd);
       LD = pow(pow(point[0],2) + point[1] - 11,2) + pow(point[0] + pow(point[1],2) - 7,2) + lambda*alpha*(gd);
       //printf("\nLD=%lf",LD);
  cont=0;
     //Armijo
      while((LE > LD)){ // || (alpha >= 0.01)  &&(cont<=12)
         alpha=alpha/2;
             LE = pow(pow(point[0]+alpha*dir[0],2) + (point[1]+alpha*dir[1]) - 11,2) + pow((point[0]+alpha*dir[0])+
              + pow(point[1]+alpha*dir[1],2) - 7,2);
         //printf("\nLE=%lf",LE);
         mult_vetores(dim,g,dir,&gd);
```

```
        LD = pow(pow(point[0],2) + point[1] - 11,2) + pow(point[0] + pow(point[1],2) - 7,2) + lambda*alpha*(gd);
        //printf("\nLD=%lf",LD);
        cont=cont+1;
         }
         //printf("\nLE=%lf",LE);
         //printf("\nLD=%lf",LD);
         //printf("\ncontador=%d\n",cont);
         //printf("\nalpha=%lf\n",alpha);

      //*********************************************************************


        //calcula novo ponto
       for(i=0;i<dim;i++){
                          point[i]=point[i]+alpha*dir[i];
                          printf("\n\n\tx[%d]= %lf",i+1,point[i]);
                          }

       //calcula novo gradiente(k+1)
       funcao_teste_desconhecida(dim,numero_funcoes,point,g,&f_x);

       //calcula Beta
       calcula_beta(dim,escolha_beta,t,g,g_anterior,&beta);

      //calcula nova direcao (k+1)
      for(j=0;j<dim;j++){
                         dir[j]=-g[j]+beta*dir_anterior[j];
                         }

//Incrementa k e t
k=k+1;
t=t+1;

  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);
```

```
   //Imprimi as informacoes no arquivo texto
   fprintf(arquivo,"\niteracao=%d\n",k);
      for(j=0;j<dim;j++){
                        fprintf(arquivo,"\nx[%d]=%1.10lf\t",j+1,point[j]);
                         }


   } // fim while

tFim = clock();


/*calcula aproximadamente o tempo em milisegundos gasto entre as duas chamadas de clock()*/
tDecorrido = ((tFim - tInicio) / (CLOCKS_PER_SEC));



           //Imprime tempo gasto para o cálculo da solucao
            printf("\n\nTempo gasto: %lf seg\n",tDecorrido);


    //Arquivando uma possivel solucao num arquivo txt
                        for(j=0;j<dim;j++){
                                          arq_ponto[j] = point[j];
                                          }

     //Imprimi Solucao
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                         printf("\t\tx[%d]=%lf",j+1,point[j]);
                         }

   for(j=0;j<dim;j++){
                        fprintf(arquivo,"\nx[%d]=%lf\t",j+1,arq_ponto[j]);
                         }

     //Valor da Funcao f(x)
     funcao_teste_desconhecida(dim,numero_funcoes,point,g,&f_x);//Recalcula f(x) no minimizador
     printf("\n\n\tf(x)=%2.13lf",f_x);
```

```
printf("\n\n*********************************************************************************\n\n");
}fclose(arquivo);

                    break;

          case 10: printf("\n*********************************************************************************\n");

  //********* Cria um arquivo txt com as solucoes *****************
arquivo = fopen("Funcao_Six_Hump_Camel_Modificada.txt", "w+t");
if(arquivo!=NULL)
{

      //Funcao Six-hump Camel Modificada
      dim=2;
      numero_funcoes=dim;

      printf("\t******Entre com o ponto inicial******\n\n");
      //**********************Dados iniciais*********************
      // ponto inicial
      for(i=0;i<dim;i++){
      printf("\tDigite o valor de x[%d]: ",i+1); ;
      scanf("%lf",&point[i]);
                        }
      printf("\n\n\tEscolha o beta [1] Fletcher-Reeves [2] Polak-Ribiere [5]LL1 [6]LL2: ");
      scanf("%d",&escolha_beta);
k=0;
norma=0;

tInicio = clock();

//Calcula o gradiente com ponto inicial
funcao_six_hump_camel_modificada(point,g);

//direcao inicial
      for(i=0;i<dim;i++){
                         dir[i]=-g[i];
                         }
      for(i=0;i<dim;i++){
```

```
                        printf("\n\tdirecao[%d]=%lf \n",i+1,dir[i]);
                        }
//************************************************************

   //criterio de parada
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\tnorma=%lf",norma);

   while(norma>=epsilon){
                        printf("\n\niteracao=%d\n",k+2); //imprimi em qual iteracao está

       //Guarda o gradiente  e direcao
       for(i=0;i<dim;i++){
                         g_anterior[i]=0; //Eliminar lixo
                         g_anterior[i]=g[i];
                         dir_anterior[i]=0; //Eliminar lixo
                         dir_anterior[i]=dir[i];
                         }

      //calcula alpha com busca linear backtracking ****************************
       alpha=1; LE=0; LD=0;
       LE = -(-4 + 2.1*pow(point[0]+alpha*dir[0],2) - (pow(point[0]+alpha*dir[0],4)/3))*pow(point[0]+alpha*dir[0],2)+
        + (point[0]+alpha*dir[0])*(point[1]+alpha*dir[1])-
          -(4 - 4*pow(point[1]+alpha*dir[1],2))*pow(point[1]+alpha*dir[1],2) ;
       //printf("\nLE=%lf",LE);
       mult_vetores(dim,g,dir,&gd);
       LD = -(4 + 2.1*pow(point[0],2) -(pow(point[0],4)/3))*pow(point[0],2)+
        + point[0]*point[1] -(4 - 4*pow(point[1],2))*pow(point[1],2) + lambda*alpha*(gd);
       //printf("\nLD=%lf",LD);
  cont=0;
     //Armijo
      while((LE > LD)&&(cont<=12)){ // || (alpha >= 0.01)  &&(cont<=12)
         alpha=alpha/2;
             LE = -(-4 + 2.1*pow(point[0]+alpha*dir[0],2)-
              - (pow(point[0]+alpha*dir[0],4)/3))*pow(point[0]+alpha*dir[0],2) + (point[0]+alpha*dir[0])*(point[1]+alpha*dir
         //printf("\nLE=%lf",LE);
         mult_vetores(dim,g,dir,&gd);
         LD = -(4 + 2.1*pow(point[0],2) -(pow(point[0],4)/3))*pow(point[0],2)+
```

```
    + point[0]*point[1] -(4 - 4*pow(point[1],2))*pow(point[1],2) + lambda*alpha*(gd);
   //printf("\nLD=%lf",LD);
   cont=cont+1;
    }
    //printf("\nLE=%lf",LE);
    //printf("\nLD=%lf",LD);
    //printf("\ncontador=%d\n",cont);
    //printf("\nalpha=%lf\n",alpha);

//*********************************************************************


  //calcula novo ponto
 for(i=0;i<dim;i++){
                   point[i]=point[i]+alpha*dir[i];
                   printf("\n\n\tx[%d]= %lf",i+1,point[i]);
                   }

 //calcula novo gradiente(k+1)
 funcao_six_hump_camel_modificada(point,g);

//calcula Beta

       if(escolha_beta==1){

      //Fletcher-Reeves
                        mult_vetores(dim,g,g,&numerador);
                        //printf("\nnumerador=%lf",numerador);
                        mult_vetores(dim,g_anterior,g_anterior,&denominador);
                        //printf("\ndenominador=%lf",denominador);
           beta=(numerador)/(denominador);
          //printf("\n\nbeta=%lf\n",beta);

                                       }
       else{

     //Polak-Ribiere
                   for(i=0;i<dim;i++){
                                      sub_g_gant[i]=0; //Eliminar possiveis lixos
```

```
                                        sub_g_gant[i]=g[i]-g_anterior[i];
                                        }
                          mult_vetores(dim,g,sub_g_gant,&numerador);
                          mult_vetores(dim,g_anterior,g_anterior,&denominador);

                  beta=(numerador)/(denominador);

                }

      //calcula nova direcao (k+1)
      for(j=0;j<dim;j++){
                         dir[j]=-g[j]+beta*dir_anterior[j];
                         }

//Incrementa k
k=k+1;

  //Calcula a norma do gradiente para saber se ja encontramos a solucao ou se estamos proximos
   mult_vetores(dim,g,g,&resul_norma);
   norma=sqrt(resul_norma);
   printf("\n\n\tnorma=%lf",norma);


   //Imprimi as informacoes no arquivo texto
   fprintf(arquivo,"\niteracao=%d\n",k);
      for(j=0;j<dim;j++){
                        fprintf(arquivo,"\nx[%d]=%1.10lf\t",j+1,point[j]);
                         }


   } // fim while

tFim = clock();


/*calcula aproximadamente o tempo em milisegundos gasto entre as duas chamadas de clock()*/
tDecorrido = ((tFim - tInicio) / (CLOCKS_PER_SEC));
```

```
          //Imprime tempo gasto para o cálculo da solucao
           printf("\n\nTempo gasto: %lf seg\n",tDecorrido);


    //Arquivando uma possivel solucao num arquivo txt
                      for(j=0;j<dim;j++){
                                        arq_ponto[j] = point[j];
                                        }

     //Imprimi Solucao
     printf("\n\n\n\t\t\tA Solucao do Problema\n\n");
     for(j=0;j<dim;j++){
                       printf("\t\tx[%d]=%lf",j+1,point[j]);
                       }

   for(j=0;j<dim;j++){
                      fprintf(arquivo,"\nx[%d]=%lf\t",j+1,arq_ponto[j]);
                       }

printf("\n\n*********************************************************************************\n\n");
}fclose(arquivo);

                   break;

        case 11: sai = 1;
                   break;

        default : printf("\nERRO! Opcao invalida.\n\n");

      }
}

} //fim rotina gradientes conjugados


int main()
{
```

```
 gradientes_conjugados();


  printf("\n\n\n");
  system("PAUSE");
return 0;
}
```