UM MÉTODO PARA A SOLUÇÃO DE SISTEMAS LINEARES ATRAVÉS DO GRADIENTE CONJUGADO COOPERATIVO

Guilherme da Silva Niedu

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Amit Bhaya

Rio de Janeiro
Março de 2012

UM MÉTODO PARA A SOLUÇÃO DE SISTEMAS LINEARES ATRAVÉS DO GRADIENTE CONJUGADO COOPERATIVO

Guilherme da Silva Niedu

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

Prof. Amit Bhaya, Ph.D.

Prof. Philippe Remy Bernard Devloo, Ph.D.

Prof. João Carlos dos Santos Basílio, D. Phil.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2012

da Silva Niedu, Guilherme

Um método para a solução de sistemas lineares através do gradiente conjugado cooperativo/Guilherme da Silva Niedu. – Rio de Janeiro: UFRJ/COPPE, 2012.

XII, 117 p.: il.; 29, 7cm.

Orientador: Amit Bhaya

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2012.

Referências Bibliográficas: p. 72 – 73.

1. Gradiente. 2. Conjugado. 3. Multitarefa. I. Bhaya, Amit. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Ao meu pai, um vitorioso na batalha contra o câncer

Agradecimentos

Gostaria de agradecer o apoio da minha família, ao meu colega de laboratório Fernando Pazos, que participou desta pesquisa e foi de grande valor para ela e ao meu orientador, Amit Bhaya, que, além de toda a ajuda, foi acima de tudo um amigo ao longo deste trabalho. Este é um trabalho feito a oito mãos, por mim, por Amit Bhaya, Fernando Pazos e Pierre-Alexandre Bliman e, embora tenha sido minha a responsabilidade da implementação e experimentação, todos tiveram a sua contribuição para essa dissertação.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UM MÉTODO PARA A SOLUÇÃO DE SISTEMAS LINEARES ATRAVÉS DO GRADIENTE CONJUGADO COOPERATIVO

Guilherme da Silva Niedu

Março/2012

Orientador: Amit Bhaya

Programa: Engenharia Elétrica

Essa dissertação explora a idéia de cooperação entre agentes (ou tarefas) no contexto do algoritmo Gradiente Conjugado (CG). Esse famoso algoritmo apresenta uma série de propriedades interessantes, tanto como algoritmo em aritmética exata quanto como em aritmética de precisão finita. No entanto, é amplamente conhecido que, devido à sua estrutura, o algoritmo CG tradicional não pode ser paralelizado de maneira comum. Neste trabalho, o algoritmo CG é revisto, de uma perspectiva multi-tarefa, que pode ser vista como uma generalização direta da abordagem de controle do algoritmo CG proposta anteriormente na literatura, na qual os parâmetros escalares de controle (= passos nas direções gradientes e gradientes conjugadas) são substituídos por matrizes. A cooperação entre agentes reside no fato dos cálculos de cada entrada da matriz de parâmetros de controle agora envolvem informação vinda dos outros agentes. O método pode também ser visto como uma generalização do algoritmo CG tradicional na qual múltiplas direções de descida e direções gradientes conjugadas são atualizadas simultaneamente. Além disso, é uma generalização que permite implementação multi-tarefa de baixa complexidade em aritmética exata. Um passo adicional de cooperação, envolvendo uma projeção afim, projetado para acelerar a computação também é discutido posteriormente. São apresentados experimentos numéricos que ilustram os resultados teóricos.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A METHOD FOR SOLVING LINEAR SYSTEMS THROUGH COOPERATIVE CONJUGATE GRADIENT

Guilherme da Silva Niedu

March/2012

Advisor: Amit Bhaya

Department: Electrical Engineering

This dissertation explores the idea of cooperation between agents (or threads) in the context of the conjugate gradient (CG) algorithm. This famous algorithm has several interesting properties, both as an algorithm in exact arithmetic and as one in finite precision arithmetic. However, it is well known that, due to its structure, the traditional CG algorithm cannot be parallelized in the conventional sense. In this work, the traditional CG algorithm is revisited from a multi-thread perspective, which can be seen as a direct generalization of the control approach to the CG algorithm proposed earlier in the literature, in which the scalar control parameters (= stepsizes in gradient and conjugate gradient directions) are replaced with matrices. The cooperation between agents resides in the fact that the calculation of each entry of the control parameter matrix now involves information that comes from the other agents. The method can also be seen as a generalization of the traditional CG algorithm in which multiple descent and conjugate directions are updated simultaneously. Moreover, it is a generalization that permits multi-thread implementation of low complexity in exact arithmetic. An additional cooperation step, involving an affine projection, designed to speed up computation further is also discussed. Numerical experiments that illustrate the theoretical results are provided.

Sumário

Lista de Figuras

Lista de Tabelas

Notação utilizada

Conjunto de vetores $r_i \in \mathbb{R}^N$, $i = 0, 1, \ldots, k$	$\{r_i\}_0^k$
Matriz obtida pela concatenação dos vetores r_i	$[r_i]_0^k$
Subespaço de combinações lineares das colunas da matriz $[r_i]_0^k$	$span[r_i]_0^k$
Subespaço de combinações lineares das colunas de $[R_i]_0^k$	$span[R_i]_0^k$
Valor de x em $Ax = b$ na k-ésima iteração	x_k
Direção de busca	d_k
Erro	e_k
Resíduo	r_k
Passo em $x_{k+1} = x_k + \alpha_k d_k$	α_k
Passo em $d_{k+1} = r_{k+1} + \beta_k d_k$	β_k

Capítulo 1

Introdução

Hoje em dia, com o aparecimento de problemas que requerem a solução numérica de equações diferenciais de dimensões cada vez mais elevadas, faz-se necessário o uso de métodos numéricos cada vez mais rápidos para resolvê-las. A maioria desses métodos resulta em sistemas lineares de dimensões bastante elevadas impedindo o uso de métodos diretos, tais como a Eliminação Gaussiana, e exigindo a utilização de métodos iterativos que sejam capazes de gerar boas aproximações da solução rapidamente.

Com a capacidade de processamento dos computadores crescendo rapidamente, e com o advento de processadores multi-núcleo, com tais núcleos num mesmo *chip*, faz-se necessário considerar o uso de processamento paralelo para melhorar tais cálculos. Por outro lado, o paradigma da cooperação entre agentes para a realização de determinada tarefa tem se tornado muito comum em áreas como controle e computação distribuída, representando várias situações e questões matemáticas associadas, conforme se observa em [1], [2] e [3].

Na área de computação, a ênfase tem sido na computação paralela, onde uma tarefa principal é dividida entre múltiplos agentes, tantos quantos forem a quantidade disponível de processadores. Essa subdivisão induz a idéia de uma estrutura de comunicação entre esses agentes, de forma a maximizar a simultaneidade entre eles, minimizando o tempo gasto para a comunicação entre agentes e, conseqüentemente, o tempo de computação total. Esse desafio de maximizar a simultaneidade entre as subtarefas surge como uma conseqüência de máquinas de múltiplos processadores mais antigas, onde a comunicação entre processadores era consideravelmente mais lenta que a execução de operações matemáticas. Uma desvantagem significativa dessa abordagem reside na dificuldade em se dividir uma tarefa em várias subtarefas que possam ser executadas simultaneamente, e no fato de serem necessárias barreiras para se forçar um ponto de sincronismo entre tarefas.

Mais recentemente, na área de controle, o interesse tem se concentrado em sistemas com múltiplos agentes, onde tais agentes cooperam entre si de forma distribuída, com o

intuito de se executar uma tarefa computacional. De forma similar, na área de computação, os processadores de múltiplos núcleos tem se tornado comuns. Nesses processadores, cada núcleo recebe uma tarefa que é executada de forma independente das tarefas nos outros núcleos, porém todos têm acesso relativamente rápido a uma memória compartilhada, tornando a comunicação entre núcleos menos onerosa em termos de tempo gasto. No contexto dessa dissertação, que é focada na resolução de um sistema linear $Ax = b$, com A sendo positiva definida e de dimensão N grande, será suposto que cada agente executará um *thread* (tarefa) em um núcleo de processamento. Assim, nesse contexto, as palavras "agente" e "*thread*" representarão o mesmo conceito. A palavra "agente" será utilizada mais comumente ao longo deste trabalho.

Com o advento de mais memória disponível, torna-se possível propor um arranjo no qual cada agente, acessando uma memória comum, calcula a sua estimativa do problema completo, evitando a divisão em subproblemas. Os agentes também trocam informação entre si através de uma memória compartilhada, sendo esse o passo de comunicação. Esse modelo de comunicação é definido como o modelo de "Quadro Negro" (*Blackboard*) em inteligência artificial [4]. Um algoritmo cooperativo deve ser idealizado de forma a se minimizar o tempo total de convergência, controlando-se as comunicações entre os agentes.

1.1 Cooperação entre algoritmos: híbridos ou times

O conceito de cooperação entre agentes, entendidos como algoritmos distintos é antigo - na comunidade de engenharia foi introduzido o conceito de algoritmo de equipe (*Team algorithm*) em [5] - , sendo utilizado na álgebra linear numérica muito antes do advento de processadores de múltiplos núcleos, por Brezinski e Redivo-Zaglia em [6], tendo sido nomeada "procedimentos híbridos" e definida como "uma combinação de duas soluções aproximadas arbitrárias com coeficientes que somam 1... (para que) a combinação somente dependa de um parâmetro cujo valor é escolhido com o intuito de minimizar a norma euclidiana do vetor resíduo obtido pelo procedimento híbrido... As duas soluções aproximadas que são combinadas num procedimento híbrido são usualmente obtidas por dois métodos iterativos."(traduzido do inglês). A idéia de se minimizar a norma do resíduo tem o objetivo de acelerar a convergência do procedimento híbrido como um todo. Essa idéia foi generalizada e discutida no contexto da computação assíncrona em [7].

1.2 Algoritmos de gradientes conjugados

No paradigma da solução de sistemas lineares de dimensão elevada, faz-se necessário o uso de métodos alternativos a inversão da matriz ou a Eliminação Gaussiana. É nesse contexto que se utiliza a otimização para obter tal solução. É proposta uma função a ser otimizada por um algoritmo, de forma que o mínimo dessa função seja a solução do sistema linear. O exemplo mais conhecido dessa classe de algoritmos é o Gradiente Conjugado (CG), desenvolvido por Hestenes e Stiefel em [8].

Entretanto, pela forma como o CG foi proposto, a sua paralelização não é possível, pois cada iteração desse algoritmo envolve uma seqüência de cálculos, onde cada cálculo utiliza informação do cálculo anterior. Dessa forma, mesmo que todas essas computações fossem distribuídas para vários agentes, seria necessário que um agente esperasse o término do cálculo do anterior para efetuar a sua computação. Em outras palavras, o algoritmo CG (da forma como proposto originalmente) é essencialmente seqüencial.

Mais especificamente, esta dissertação explora a idéia da cooperação entre agentes (ou *threads*) no contexto ao algoritmo Gradiente Conjugado (CG), aplicado a um sistema $Ax = b$ de dimensão N, com A sendo simétrica positiva definida, de dimensão N elevada. Sendo utilizados P agentes, supõe-se que $P \ll N$, isto é, o número de agentes é pequeno quando comparado à dimensão do problema. O famoso algoritmo Gradiente Conjugado apresenta uma série de propriedades interessantes, tanto em aritmética exata como em aritmética de precisão finita, conforme visto em [9] e [10].

O algoritmo CG é da classe de métodos de subespaço Krylov, conforme visto em [11, pp. 300-321] e em [12, pp. 245-248]. Esses métodos pressupõem que não há acesso direto à matriz A, e que apenas se pode fornecer um vetor x e conhecer a resposta Ax (ou $A^T x$, se A for não simétrica), sem se poder manipular ou ter acesso à matriz em si. Essa implicação é importante pois a operação não trivial menos custosa que se pode fazer com uma matriz é multiplicá-la por um vetor. É possível extrair informação a respeito de A e resolver $Ax = b$ utilizando-se subespaços de Krylov, que são os espaços gerados pelos vetores $\{b, Ab, \ldots, A^{k-1}b\}$, conforme citado em [11, pp. 300-321]. Devido à forma como será proposto, o cCG também se encaixa nessa classe de métodos.

Dessa forma, nessa dissertação, o algoritmo CG é revisitado de uma perspectiva multitarefa, que pode ser vista como uma generalização direta da abordagem de controle proposta ao CG em [13, pp. 77-82], no qual os parâmetros escalares de controle (passos no algoritmo CG) são substituídos por matrizes (controle multivariável). Esse método pode também ser visto como uma generalização do algoritmo CG onde múltiplas direções de busca conjugadas são atualizadas simultaneamente. O novo algoritmo proposto foi nomeado Gradiente Conjugado Cooperativo (cCG), e nele o problema básico do CG

é distribuído por múltiplos agentes (*threads*). O conceito desenvolvido para o cCG é o de cooperação (e não o de paralelismo convencional), isto é, cada um dos agentes (*threads*) tem acesso ao problema completo (ao contrário do paralelismo, onde o problema é dividido) e troca informações com os demais agentes.

A principal diferença entre o método proposto por Brezinski e Redivo-Zaglia em [6] e o proposto neste trabalho é que a comunicação não envolve as soluções computadas, mas sim as direções de busca. Isto é, cada agente terá a sua própria solução computada, e essa solução será atualizada utilizando-se a informação de todas as P direções de busca geradas em uma determinada iteração. O algoritmo proposto por Brezinski e Redivo-Zaglia fazia a suposição de dois agentes, de forma a se ter que calcular apenas um parâmetro (já que os parâmetros totalizam 1) de forma a se minimizar a norma do resíduo das soluções computadas. O contexto proposto aqui é diferente, embora uma adaptação desse contexto (combinação linear com pesos que somam 1) como um passo adicional ao cCG também seja proposta nesta dissertação a posteriori, na seção 5.

1.3 Complexidade do algoritmo

Como uma análise da complexidade do algoritmo, além de se medir a quantidade de iterações ou o tempo gastos para convergência, pode-se medir a quantidade de acessos à matriz A (isto é, o número de vezes em que se fornece um vetor e se obtém o produto de A por esse vetor). Essa metodologia é proposta em [14] e, embora não seja exatamente o que é computado nesta dissertação, a mesma considera a complexidade de métodos de Krylov em termos de multiplicações matriz-vetor dentro de uma certa tolerância, o que é o enfoque deste trabalho. A metodologia utilizada nessa dissertação para a análise da complexidade do cCG em comparação com o CG é a quantidade de operações entre escalares realizadas por cada agente.

1.4 Organização da dissertação

Este trabalho está dividido de forma a explicar, inicialmente, o conceito de otimização para solução de sistemas lineares. Em seguida, é abordado o algoritmo CG original, com breve explicações sobre seu funcionamento sua convergência. Com as bases de cálculo do CG, é proposto o algoritmo principal deste trabalho, o cCG, e são discutidos os seus modelos de operações realizadas e rapidez de convergência. No capítulo seguinte é proposto um passo adicional ao cCG para torná-lo ainda mais rápido. Por fim, são feitos múltiplos experimentos numéricos a fim de ilustrar e validar as hipóteses propostas e conclusões a respeito dos mesmos são tiradas.

Capítulo 2

Otimização para solução de sistemas lineares

Ultimamente a necessidade de encontrar soluções de sistemas lineares tem se tornado cada vez mais relevante. Tais problemas surgem em diversas áreas do conhecimento, sobretudo em aproximações numéricas para solução de problemas. Um exemplo clássico desses problemas são as Equações Diferenciais Parciais que, ao serem resolvidas por métodos numéricos, geram sistemas lineares de dimensão elevada.

Um sistema linear na forma de um produto matricial pode ser visto como

$$Ax = b \tag{2.1}$$

sendo $A \in \mathbb{R}^{N \times N}$, $x \in \mathbb{R}^N$ e $b \in \mathbb{R}^N$, e os vetores são tomados como vetores coluna. Nesse sentido, a solução do sistema pode ser calculada de diversas maneiras. Uma delas (supondo que A seja inversível, caso contrário o sistema teria infinitas soluções ou nenhuma solução) é calcular a inversa da matriz A, isto é

$$x = A^{-1}b \tag{2.2}$$

Porém, o cálculo da inversa apresenta em geral certa imprecisão numérica, além de ser um algoritmo de ordem $O(N^3)$, isto é, a quantidade de operações de multiplicação entre escalares cresce com o cubo da dimensão do problema. Uma outra solução seria a eliminação Gaussiana, um método que apresenta maior precisão, mas ainda um método de ordem $O(N^3)$.

Sendo assim, faz-se necessário lançar mão de outros recursos para a solução de problemas de dimensão elevada, tais como N da ordem de milhares ou mesmo milhões. É nesse contexto que é utilizada a otimização para a solução do problema visto em (2.1). Portanto, propõe-se a seguinte função de otimização

$$f(x) = \frac{x^T Ax}{2} - b^T x \tag{2.3}$$

Ao derivarmos (2.3) em relação ao vetor x produziremos um vetor também pertencente a $\mathbb{R}^N$. Arranjando esse vetor na forma de um vetor coluna obteremos a expressão a seguir.

$$\nabla f(x) = \frac{A + A^T}{2} x - b \tag{2.4}$$

Aqui fazemos a primeira restrição ao uso dos métodos de otimização propostos neste trabalho. Supondo que tenhamos uma matriz simétrica, isto é, com $A = A^T$, (2.4) se reduz a

$$\nabla f(x) = Ax - b \tag{2.5}$$

ou seja, a derivada da função de otimização proposta apresenta um zero exatamente na solução do sistema. Entretanto, isso não basta para que a solução do sistema possa ser encontrada por um método de otimização. Para que isso ocorra, é importante que a matriz A tenha certas características que determinarão a natureza desse zero de sua derivada.

Primeiramente, devido à simetria de A, todos os seus autovalores serão reais. Isso nos leva a classificar A com relação aos mesmos. Se todos os autovalores de A forem positivos, dizemos que A é uma matriz **Positiva Definida (PD)**. Se todos os autovalores de A forem negativos, dizemos que A é **Negativa Definida (ND)**. Se os autovalores de A forem todos maiores ou iguais a zero ou todos menores ou iguais a zero, classificamos A como **Positiva Semi-Definida (PSD)** ou **Negativa Semi-Definida (NSD)**, respectivamente. Por fim, se A tiver autovalores positivos e negativos, classificamos A como indefinida.

De acordo com a natureza de A, a função definida em (2.3) terá diferentes características para o zero de sua derivada. Se A for uma matriz PSD ou NSD teremos que o seu gráfico será como o de uma calha, com uma linha (caso $N = 2$) ou hiperplano (caso $N > 2$), conforme visto nas figuras 2.1 e 2.2, para o caso $N = 2$.

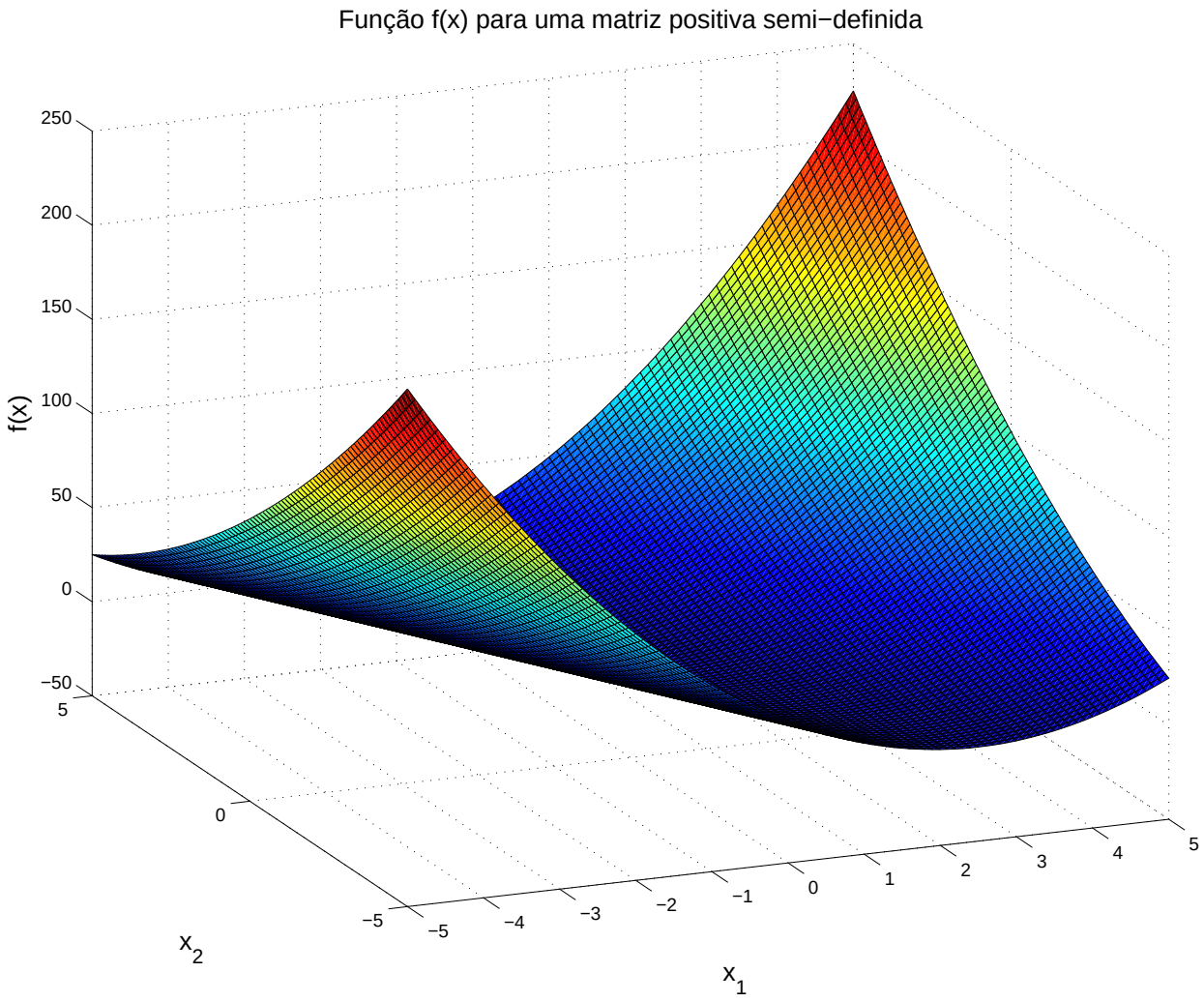

Figura 2.1: Função $f(x)$ para uma matriz PSD

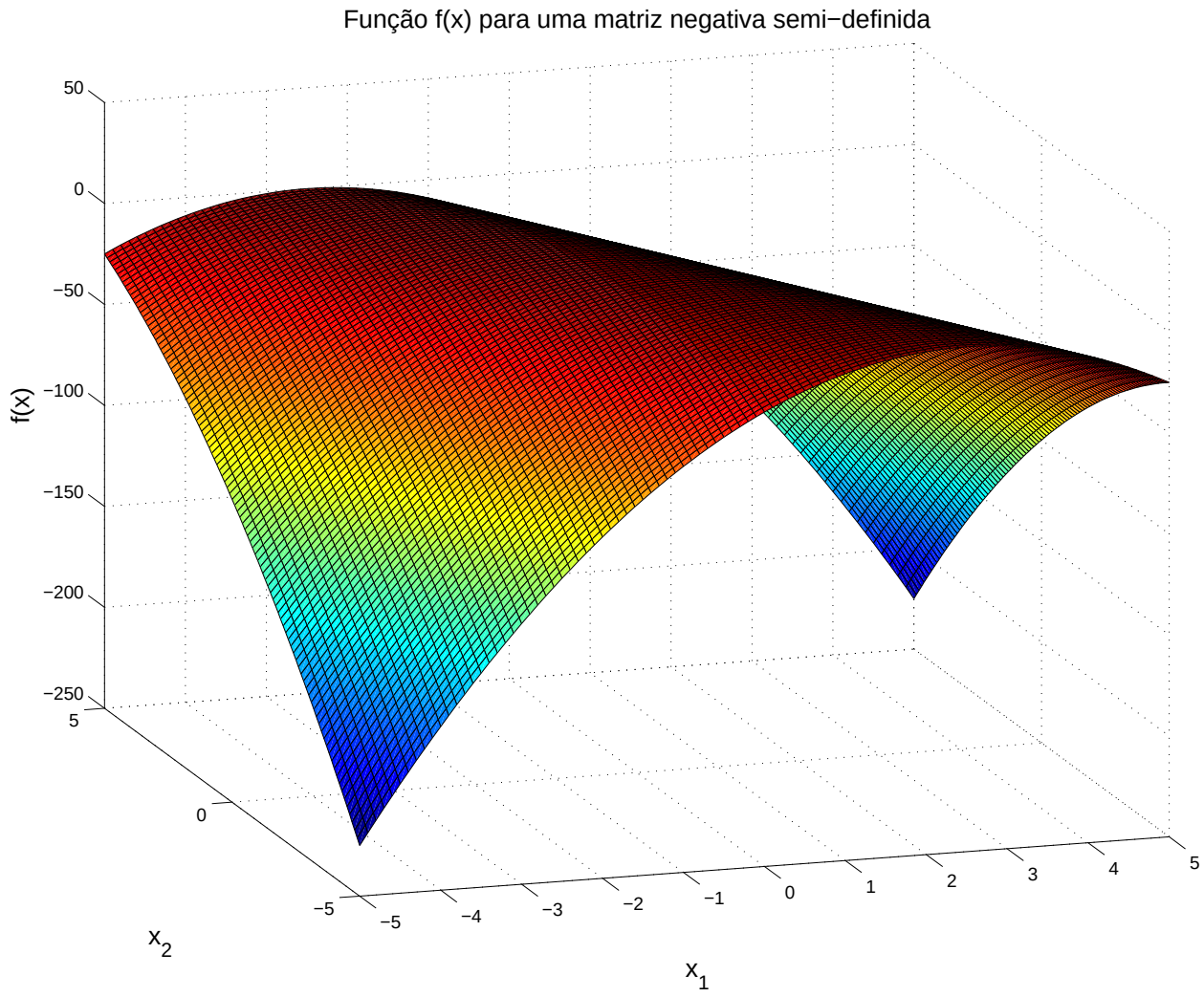

Figura 2.2: Função $f(x)$ para uma matriz NSD

Entretanto, se A for PSD ou NSD, significa que essa matriz tem pelo menos um autovalor em zero. Como o determinante de uma matriz é o produto de seus autovalores, esse determinante será zero e, portanto, A será não inversível. Conforme já foi citado, o sistema proposto em (2.1) terá solução única se e somente se A for inversível.

Se A for indefinida o zero de sua derivada não será um máximo nem um mínimo, mas sim um chamado Ponto de Sela, e algoritmos de otimização não irão convergir para ele. Um exemplo de um ponto de sela para $N = 2$ pode ser visto na figura 2.3.

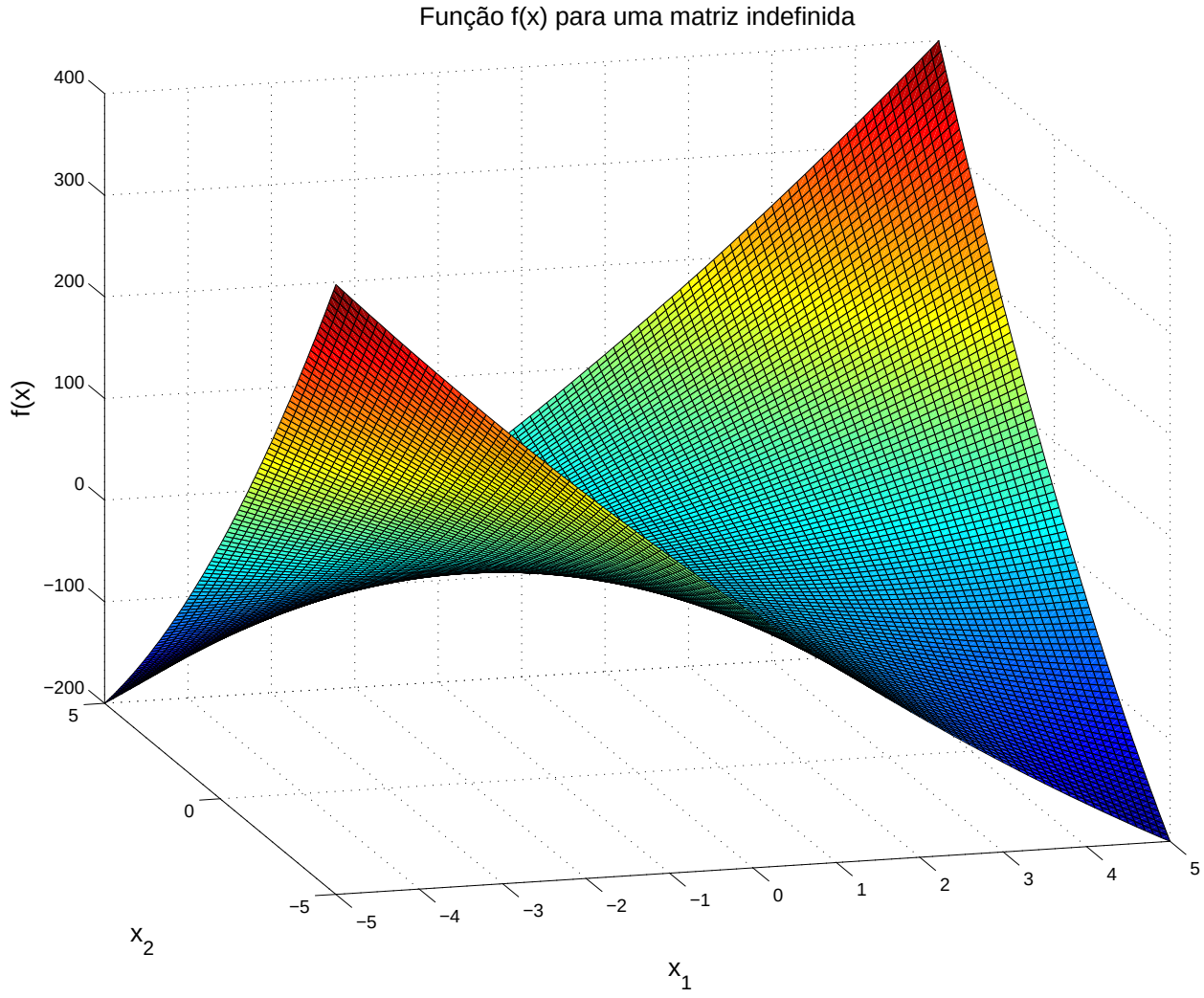

Figura 2.3: Função $f(x)$ para uma matriz indefinida

Por esse motivo, métodos de otimização baseados na minimização de (2.3) se restringem a matrizes PD e ND. Se A for Positiva Definida, o zero de sua derivada será um ponto de mínimo, e o algoritmo será capaz de convergir para ele. No entanto, se A for

negativa definida, o zero da derivada de (2.3) será um máximo. A função definida em (2.3) para matrizes PD e ND pode ser vista nas figuras 2.4 e 2.5.

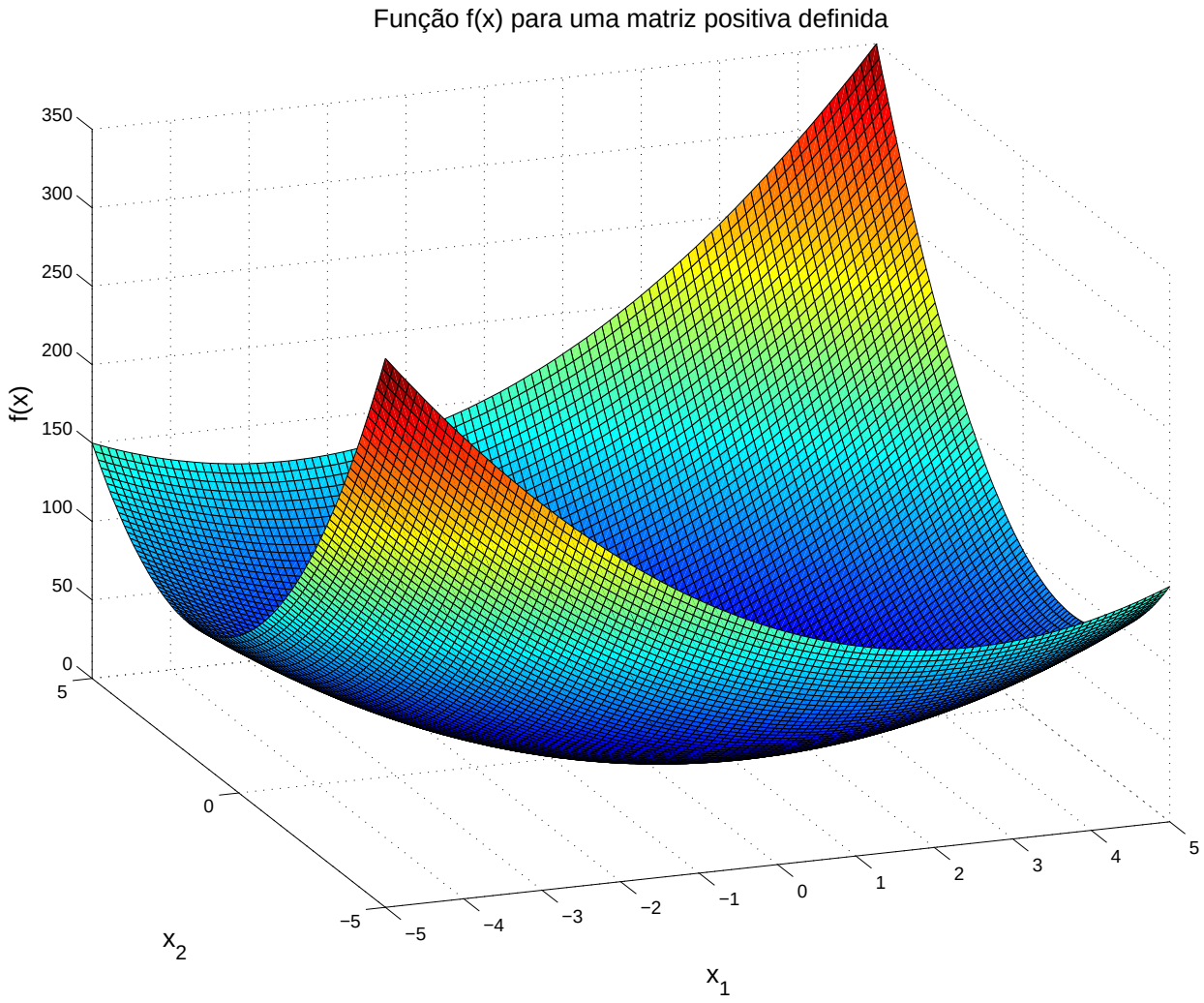

Figura 2.4: Função $f(x)$ para uma matriz PD

Portanto, as matrizes de interesse para este trabalho devem atender a seguinte hipótese.

- *Hipótese: A é uma matriz simétrica positiva definida.*

Se a matriz for negativa definida, uma simples troca de sinal a torna apta para o algoritmo de minimização. É possível fazer $C = -A$ e resolver $Cy = b$, e por fim achar a solução $x = -y$. Dessa maneira, este trabalho é focado em encontrar a solução de problemas como o de (2.1) para matrizes A positivas definidas. Os algoritmos utilizados são explicados nos capítulos a seguir.

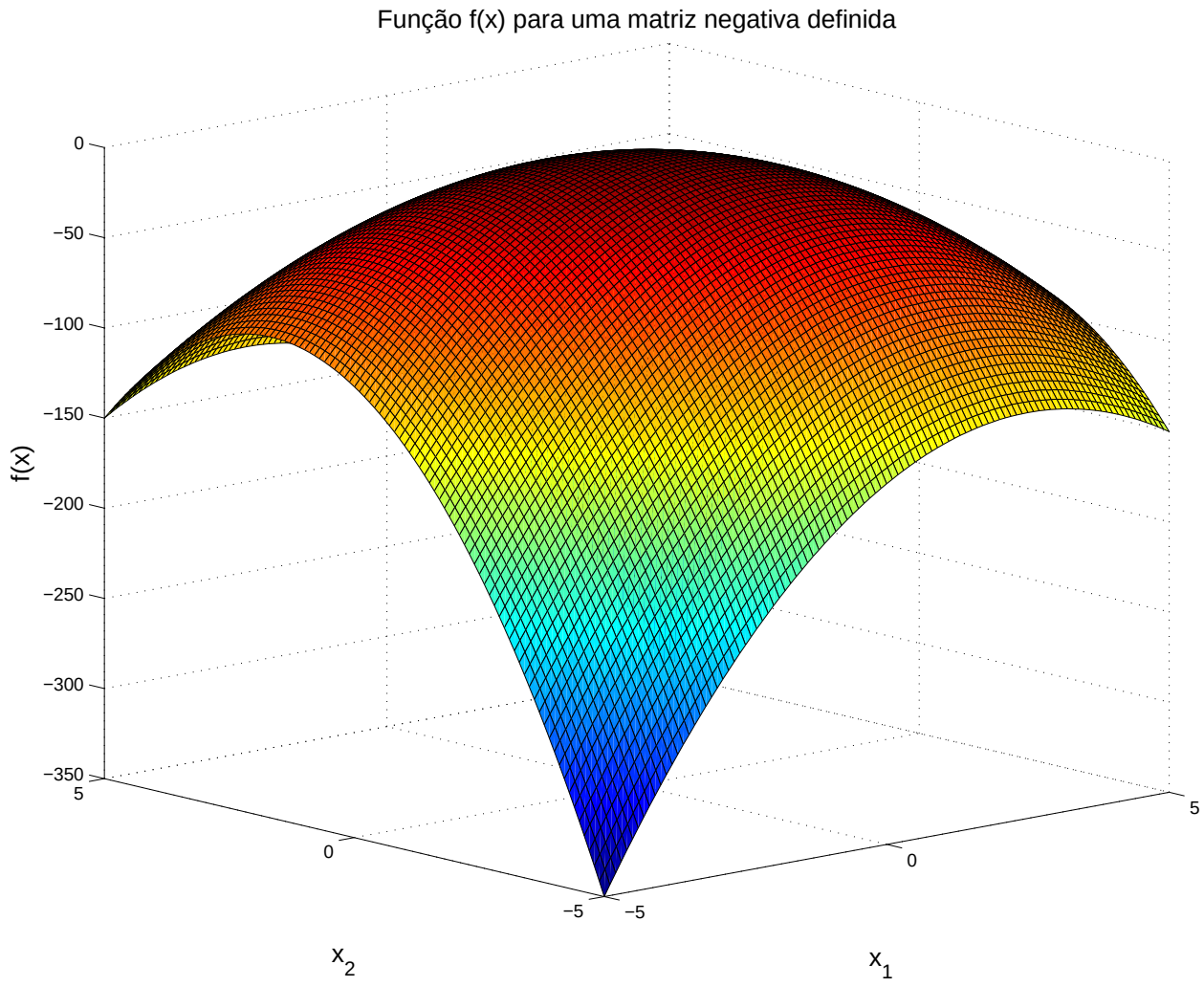

Figura 2.5: Função $f(x)$ para uma matriz ND

Capítulo 3

O método Gradiente Conjugado

Nesse capítulo, será abordada a forma como funciona o algoritmo Gradiente Conjugado (CG) clássico, proposto por Hestenes e Stiefel em [8] e mostrado de forma didática por Shewchuk em [15].

3.1 Construção do método

Um raciocínio lógico para um método de otimização a fim de resolver a solução de um problema linear seria olhar para o erro da solução, conforme definido a seguir

$$e = x - x^* = x - A^{-1}b \tag{3.1}$$

Utilizando o erro, pode ser proposto um esquema iterativo escolhendo-se a direção de busca d_k e o passo α_k de forma a minimizar o erro a cada iteração, qual seja:

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

Do ponto de vista do erro, o esquema de (3.2) se resume a

$$e_{k+1} = e_k + \alpha_k d_k \tag{3.3}$$

Uma possível idéia é escolher α_k de forma a anular uma componente do erro em cada iteração. Para isso, deve-se fazer com que o erro da iteração seguinte seja ortogonal a direção atual, isto é, que a componente nessa direção seja completamente zerada, conforme mostrado na figura 3.1, partindo do ponto x_0, em azul até o ponto x^*, em verde. Para tal, teremos que escolher α_k da seguinte forma:

$$
\begin{aligned}
d_k^T e_{k+1} &= 0 \\
d_k^T (e_k + \alpha_k d_k) &= 0 \\
\alpha_k &= -\frac{d_k^T e_k}{d_k^T d_k}
\end{aligned} \tag{3.4}
$$

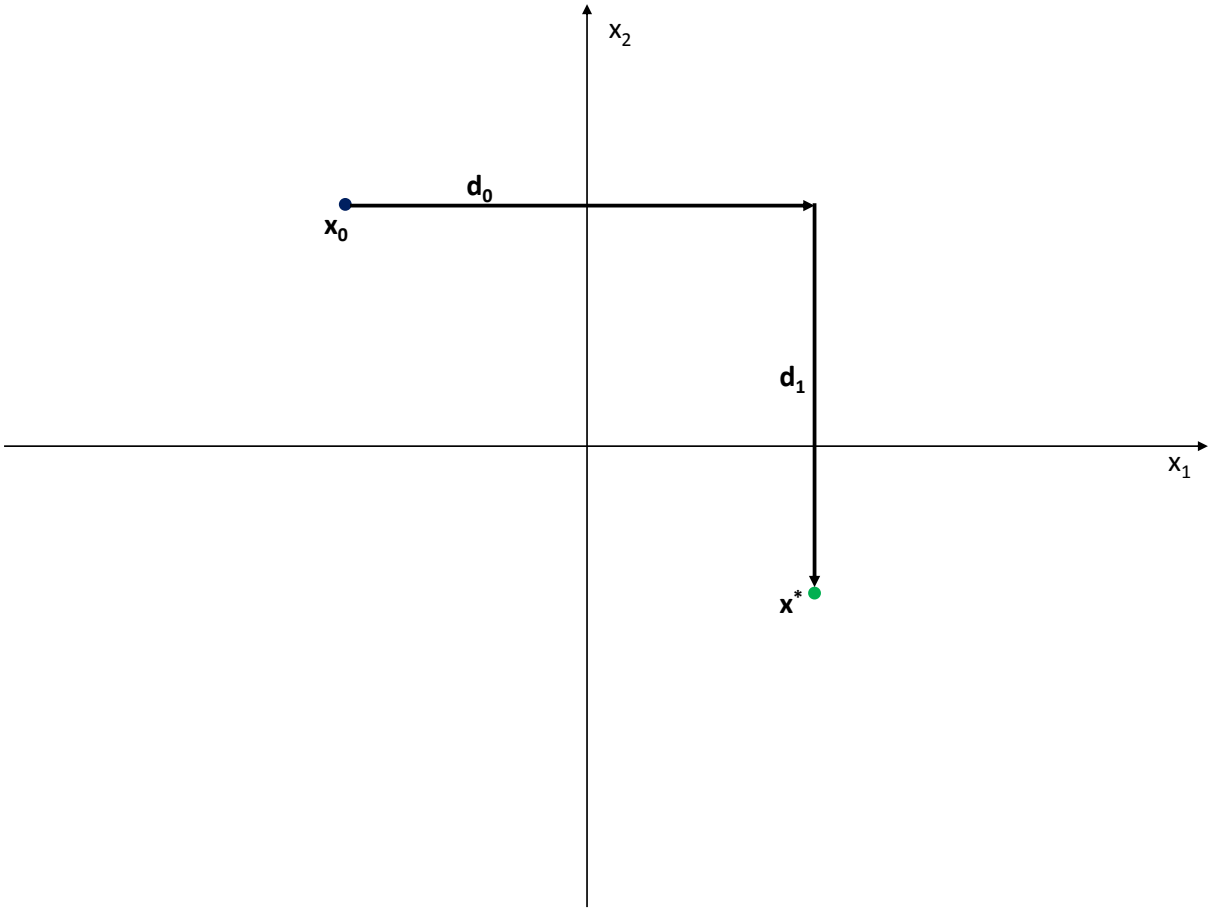

Figura 3.1: Caso ideal de otimização

Observa-se que para o cálculo de α_k é utilizada informação sobre o erro e, para conhecer o erro, é necessário conhecer a inversa de A, conforme mostrado em (3.1). Porém, o propósito de se usar um método de otimização para a solução de um sistema linear é justamente não inverter a matriz. Em outras palavras, é necessário conhecer a resposta para poder utilizar esse método.

Como uma alternativa para essa proposta, ao invés de exigirmos que as direções sejam ortogonais às direções das iterações anteriores, podemos exigir que as direções sejam A-**ortogonais** às direções das iterações anteriores. Duas direções u e v são ditas A-**ortogonais** ou **conjugadas** se elas atendem à seguinte condição:

$$u \perp_A v \quad se \quad u^T A v = 0 \tag{3.5}$$

Dessa maneira, ao invés de calcularmos α_k conforme (3.4), calculamos de modo que a direção atual e o erro da iteração seguinte sejam A-ortogonais. Esse cálculo está exposto a seguir:

$$\begin{aligned} d_k^T A e_{k+1} &= 0 \\ d_k^T A(e_k + \alpha_k d_k) & \\ \alpha_k &= -\frac{d_k^T A e_k}{d_k^T A d_k} \end{aligned} \tag{3.6}$$

A expressão para α_k exposta em (3.6) ainda usa informação sobre o erro, isto é, sobre a inversa de A. Porém, utilizando o resíduo,

$$r = Ax - b \tag{3.7}$$

e utilizando a seguinte relação,

$$\begin{aligned} r &= Ax - b \\ r &= A(x - A^{-1}b) \\ r &= Ae \end{aligned} \tag{3.8}$$

podemos chegar a uma nova expressão para α_k, mostrada a seguir.

$$\alpha_k = -\frac{d_k^T r_k}{d_k^T A d_k} \tag{3.9}$$

Em (3.9) é possível perceber que não há uso da inversa de A. A idéia de se utilizar o resíduo da iteração seguinte (r_{k+1}) ortogonal à direção atual d_k já é conhecido, conforme visto em [16]. Para mostrar que a solução x^* é computada em N passos (lembrando que $x \in \mathbb{R}^N$), podemos escrever o erro inicial (e_0) em função das direções de busca. É possível produzir N direções conjugadas linearmente independentes, de forma que as mesmas formem uma base para o $\mathbb{R}^N$ e, portanto, qualquer vetor do $\mathbb{R}^N$ possa ser escrito em função delas. Essa forma de expressar o erro inicial é observada a seguir.

$$e_0 = \sum_{i=0}^{N-1} \gamma_i d_i \tag{3.10}$$

Sendo os valores de γ_i as componentes do erro inicial em cada uma das direções de busca. Considerando que cada uma das direções de busca é A-ortogonal em relação às outras $N-1$ direções, podemos multiplicar (3.10) por $d_k^T A$ à esquerda e obter os valores de γ_k conforme vemos a seguir.

$$\begin{aligned} d_k^T A e_0 &= \sum_{i=0}^{N-1} \gamma_i d_k^T A d_i \\ d_k^T A e_0 &= \gamma_k d_k^T A d_k \\ \gamma_k &= \frac{d_k^T A e_0}{d_k^T A d_k} \end{aligned} \tag{3.11}$$

Mais uma vez, como as direções de busca são A-ortogonais, podemos somar termos contendo $d_k^T A d_j$, já que esse termo é igual a 0. Utilizando-se o fato de que

$$e_k = e_0 + \sum_{i=0}^{k-1} \alpha_i d_i \tag{3.12}$$

podemos obter o valor final de γ_k. Esse procedimento é explicitado a seguir.

$$\begin{aligned} \gamma_k &= \frac{d_k^T A(e_0 + \sum_{i=0}^{k-1} \alpha_i d_i)}{d_k^T A d_k} \\ \gamma_k &= \frac{d_k^T A e_k}{d_k^T A d_k} \\ \gamma_k &= \frac{d_k^T r_k}{d_k^T A d_k} \end{aligned} \tag{3.13}$$

Dessa forma, concluímos que para zerar o erro inicial, devemos subtrair do vetor inicial o valor de $\gamma_k d_k$, isto é, somar $\alpha_k d_k$, já que $\alpha_k = -\gamma_k$, conforme vemos por (3.13) e (3.9). Assim, após N iterações, o valor do erro e_N é zero, ou seja, em N iterações esse método chega a sua solução do problema.

Uma outra forma de determinar α_k é calcular o seu valor de forma a minimizar a norma-A do erro. Definimos uma norma ponderada por uma matriz (norma-A, desde que a mesma seja positiva definida) conforme vemos a seguir, pela definição encontrada em [17].

$$||e_k||_A = (e_k^T A e_k)^{\frac{1}{2}} \tag{3.14}$$

Como não é possível usar informação sobre o erro, deve-se usar informação sobre o resíduo. Para tal, usamos a relação definida em (3.8).

$$||e_k||_A = (e_k^T A e_k)^{\frac{1}{2}} = (r_k^T A^{-1} r_k)^{\frac{1}{2}} = ||r_k||_{A^{-1}} \tag{3.15}$$

Então, em (3.14), escolhe-se o valor de α_k que minimiza a norma-A^{-1} do resíduo na próxima iteração. Como esse valor é o mesmo que minimiza a norma ao quadrado, partimos da norma ao quadrado, derivamos em relação a α_k e igualamos a zero.

$$\begin{aligned} r_{k+1} &= Ae_{k+1} = r_k + \alpha_k A d_k \\ \|r_{k+1}\|^2_{A^{-1}} = r_k^T A^{-1} r_k + 2\alpha_k r_k^T d_k + \alpha_k^2 d_k^T A d_k \frac{\partial ||r_{k+1}||^2_{A^{-1}}}{\partial \alpha_k} &= 0 \\ 2r_k^T d_k + 2\alpha_k d_k^T A d_k &= 0 \\ \alpha_k &= -\frac{d_k^T r_k}{d_k^T A d_k} \end{aligned} \tag{3.16}$$

Uma vez que é possível achar a solução em N iterações, minimizando em cada passo ao longo de direções conjugadas, só é preciso encontrar tais direções. Para tal, é possível gerar N vetores conjugados a partir de N vetores linearmente independentes u_k, através do **Processo de Conjugação de Gram-Schmidt**. Para tanto, fazemos $d_0 = u_0$ e calculamos os demais $N-1$ vetores da seguinte forma.

$$d_k = u_k + \sum_{i=0}^{k-1} \beta_{k,i} d_i \tag{3.17}$$

Para encontrar $\beta_{k,i}$ também transpomos (3.17) e multiplicamo-na à direita por Ad_j, obtendo:

$$\begin{aligned} d_k^T A d_j &= u_k^T A d_j + \sum_{i=0}^{k-1} \beta_{k,i} d_i^T A d_j \\ 0 &= u_k^T A d_j + \beta_{k,j} d_j^T A d_j \\ \beta_{k,j} &= -\frac{u_k^T A d_j}{d_j^T A d_j} \end{aligned} \tag{3.18}$$

No entanto, ao manter os vetores u_k na memória o algoritmo se torna $O(N^3)$, assim como a inversão da matriz ou a Eliminação Gaussiana. Se os vetores u_k forem iguais aos vetores unitários correspondentes aos eixos cartesianos, o método se torna a Eliminação Gaussiana [8]. Para tornar esse método vantajoso, deve-se contornar esse fato.

Uma solução para esse problema é armazenar somente um vetor por vez. Podemos notar, escrevendo e_j em termos das componentes nas direções d_k e multiplicando a esquerda por $d_k^T A$, que o resíduo na iteração k é ortogonal a todas as direções de busca

anteriores.

$$\begin{aligned} e_j &= \sum_{i=0}^{N-1} \gamma_j d_j \\ d_k^T A e_j &= \sum_{i=0}^{N-1} \gamma_j d_k^T A d_j \\ d_k^T r_j &= 0, k < j \end{aligned} \tag{3.19}$$

Dessa maneira, escolhe-se a primeira direção de busca d_0 como sendo igual ao primeiro resíduo, r_0. A partir daí, calcula-se o valor do novo resíduo r_{k+1}, e então o valor de β (adaptando (3.18)

$$\beta_{k,j} = -\frac{r_k^T A d_j}{d_j^T A d_j} \tag{3.20}$$

com $u_k = r_{k+1}$). Com β, calcula-se a nova direção, que será garantidamente A-ortogonal à anterior (novamente adaptada de (3.17) com $u_k = r_k$).

$$d_{k+1} = r_{k+1} + \beta_k d_k, \beta_k = \beta_{k+1,k} \tag{3.21}$$

Para conferir o resultado de (3.21), podemos obter o valor de β_k a partir de (3.21), fazendo com que d_{k+1} seja A-ortogonal a d_k.

$$\begin{aligned} d_{k+1}^T A d_k &= 0 \\ (r_{k+1}^T + \beta_k d_k^T) A d_k &= 0 \\ \beta_k &= \frac{-r_{k+1}^T A d_k}{d_k^T A d_k} \end{aligned} \tag{3.22}$$

Com isso, vemos que o valor encontrado para β_k em (3.22) está perfeitamente de acordo com o valor visto em (3.20).

Uma vez definida uma forma de zerar o erro ao longo das direções de busca e uma maneira de gerar tais direções, podemos definir o método Gradiente Conjugado (CG), criado por Hestenes e Stiefel [8]. O esquema iterativo pode ser lido no pseudo-código do algoritmo 3.1.

Algoritmo 3.1 Algoritmo Gradiente Conjugado

```
choose x_0 ∈ R^N
r_0 := Ax_0 − b
d_0 := r_0
k := 0
while d_k ≠ 0 do
    α_k := −r_k^T d_k/(d_k^T A d_k)
    x_{k+1} := x_k + α_k d_k
    r_{k+1} := Ax_{k+1} − b
    β_k := −r_{k+1}^T A d_k/(d_k^T A d_k)
    d_{k+1} := r_{k+1} + β_k d_k
    k ← k + 1
end while
```

3.2 Complexidade do algoritmo

Analisando cada uma das etapas de cálculo do algoritmo, é possível fazer um levantamento da quantidade de operações realizadas em cada iteração. Serão consideradas apenas as multiplicações entre escalares, já que o tempo gasto com as somas é significativamente menor. O detalhamento das operações feitas é observado na tabela 3.1.

Tabela 3.1: Operações feitas pelo algoritmo Gradiente Conjugado

Operação	**Multiplicações entre escalares**
Ad_k	N^2
$d_k^T Ad_k$	N
$r_k^T d_k$	N
$\alpha_k := -\frac{r_k^T d_k}{(d_k^T Ad_k)}$	1
$x_{k+1} := x_k + \alpha_k d_k$	N
$r_{k+1} := r_k + \alpha_k Ad_k$	N
$-r_{k+1}^T Ad_k$	N
$\beta_k := -\frac{r_{k+1}^T Ad_k}{(d_k^T Ad_k)}$	1
$d_{k+1} := r_{k+1} + \beta_k d_k$	N
Total	$N^2 + 6N + 2$

Podemos notar que o algoritmo realiza $O(N^2)$ operações por iteração. Uma vez que para zerar o erro são gastas no máximo N iterações, conforme visto anteriormente, seria possível afirmar que o algoritmo realiza $O(N^3)$ operações. No entanto, ao se programar tal algoritmo num computador, o mesmo se transforma num esquema iterativo. Tal fato é devido à precisão de cálculo do computador ser limitada, sem haver a garantia aritmética de um máximo de N iterações para zerar o erro.

Uma vez que não há essa garantia, deve-se realizar iterações não mais até que as

direções de busca se anulem, mas sim até que um critério de parada seja atingido. Dessa maneira, o algoritmo acaba por convergir em menos de N iterações. Uma sugestão de critério de parada é fazer com que a norma do resíduo seja menor que um valor pré-determinado (uma tolerância), e um exemplo dessa implementação pode ser observado no algoritmo 3.2. Algumas propriedades do CG podem ser observadas no teorema 3.1. A dedução dessa expressão pode ser observada com mais detalhes em [18] e em [16].

Teorema 3.1 (Propriedades do CG)**.** *Enquanto o vetor d_k não é zero*

- *Os vetores $\{r_i\}_0^k$ são mutuamente ortogonais, os vetores $\{d_i\}_0^k$ são mutuamente A-ortogonais, e os subespaços $span[r_i]_0^k, span[d_i]_0^k$ e $span[A^i r_0]_0^k$ são iguais e tem dimensão $(k+1)$;*
- *O ponto x_{k+1} é o ponto que minimiza $f(x_k) = \frac{x_k^T A x_k}{2} - b^T x_k$ no subespaço afim $x_0 + span\ [d_i]_0^k$.*

Quando o vetor resíduo é zero o ótimo é atingido, mostrando que o algoritmo CG termina em tempo finito. A fórmula do teorema 3.2, retirada de [11], mostra a convergência do CG dadas k iterações, sendo κ o número de condicionamento da matriz A.

Teorema 3.2 (Propriedades do CG)**.**

- *A convergência do CG piora com o aumento da condição κ pela fórmula*

$$\|x_k - x^*\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|x_0 - x^*\|_A \tag{3.23}$$

Algoritmo 3.2 Algoritmo Gradiente Conjugado (Implementação prática)

```
choose x_0 ∈ R^N
r_0 := Ax_0 − b
d_0 := r_0
k := 0
define tol                          ▷ tolerância para critério de parada
while ||r_k|| > tol do
    α_k := −r_k^T d_k/(d_k^T A d_k)
    x_{k+1} := x_k + α_k d_k
    r_{k+1} := Ax_{k+1} − b
    β_k := −r_{k+1}^T A d_k/(d_k^T A d_k)
    d_{k+1} := r_{k+1} + β_k d_k
    k ← k + 1
end while
```

Por tal razão, o algoritmo CG é amplamente utilizado e o que apresenta o melhor desempenho para soluções de sistemas lineares por otimização. Conforme já foi mostrado,

a norma de cada resíduo a cada iteração é garantidamente mínima em norma-A^{-1}. Para ilustrar tal fato, podemos analisar a convergência para uma matriz com $N = 1000$, condição igual a 10^6 e usando como critério de parada a norma euclidiana do resíduo ser menor que 10^{-3}.

Nas figuras 3.2 e 3.3 podemos ver, respectivamente, o logaritmo da norma A^{-1} do resíduo versus a dimensão do problema e o logaritmo da norma euclidiana versus a dimensão do problema (a escala logarítmica foi escolhida pelo fato de a convergência ser muito rápida, provocando uma queda acentuada e de difícil observação em escala normal).

Podemos ver que a norma-A^{-1} do resíduo apresenta uma queda monotônica, isto é, seu valor nunca torna a subir. Tal fato é esperado, uma vez que a escolha do passo α_k minimiza a supracitada norma a cada iteração. A queda da norma euclidiana não é monotonicamente decrescente, mas podemos ver que a convergência é igualmente rápida. O critério de parada foi atingido em 269 iterações.

Observando os fatos supracitados, é possível perceber que o algoritmo CG possui alta taxa de convergência e é capaz de encontrar a solução de um sistema linear com uma elevada precisão, vencendo os demais algoritmos de otimização na solução de sistemas lineares que apresentam um tempo de convergência maior que o CG. No entanto, é possível tornar o CG ainda mais rápido. Esse assunto será abordado nas próximas seções.

3.3 O CG enquanto sistema de controle

Uma outra perspectiva para o algoritmo CG é observá-lo enquanto um sistema de controle. Essa abordagem é feita por [13] e [19]. É possível pensar no esquema de atualização de soluções computadas, visto em (3.2) de uma forma diferente. Nessa abordagem, os passos α_k e β_k serão como ganhos de um sistema de controle. O algoritmo CG pode ser deduzido a partir do método *steepest descent* clássico, cujo esquema iterativo é observado a seguir.

$$x_{k+1} = x_k + \alpha_k[r_k + \gamma_k(x_k - x_{k-1})] \tag{3.24}$$

que pode ser reescrito da seguinte forma:

$$x_{k+1} = x_k + \alpha_k d_k, \text{em que} \tag{3.25}$$

$$d_k = r_k + \gamma_k(x_k - x_{k-1}) = r_k + \gamma_k\alpha_{k-1}d_{k-1} = r_k + \beta_{k-1}d_{k-1} \tag{3.26}$$

Por (3.25) e (3.26) chegamos a

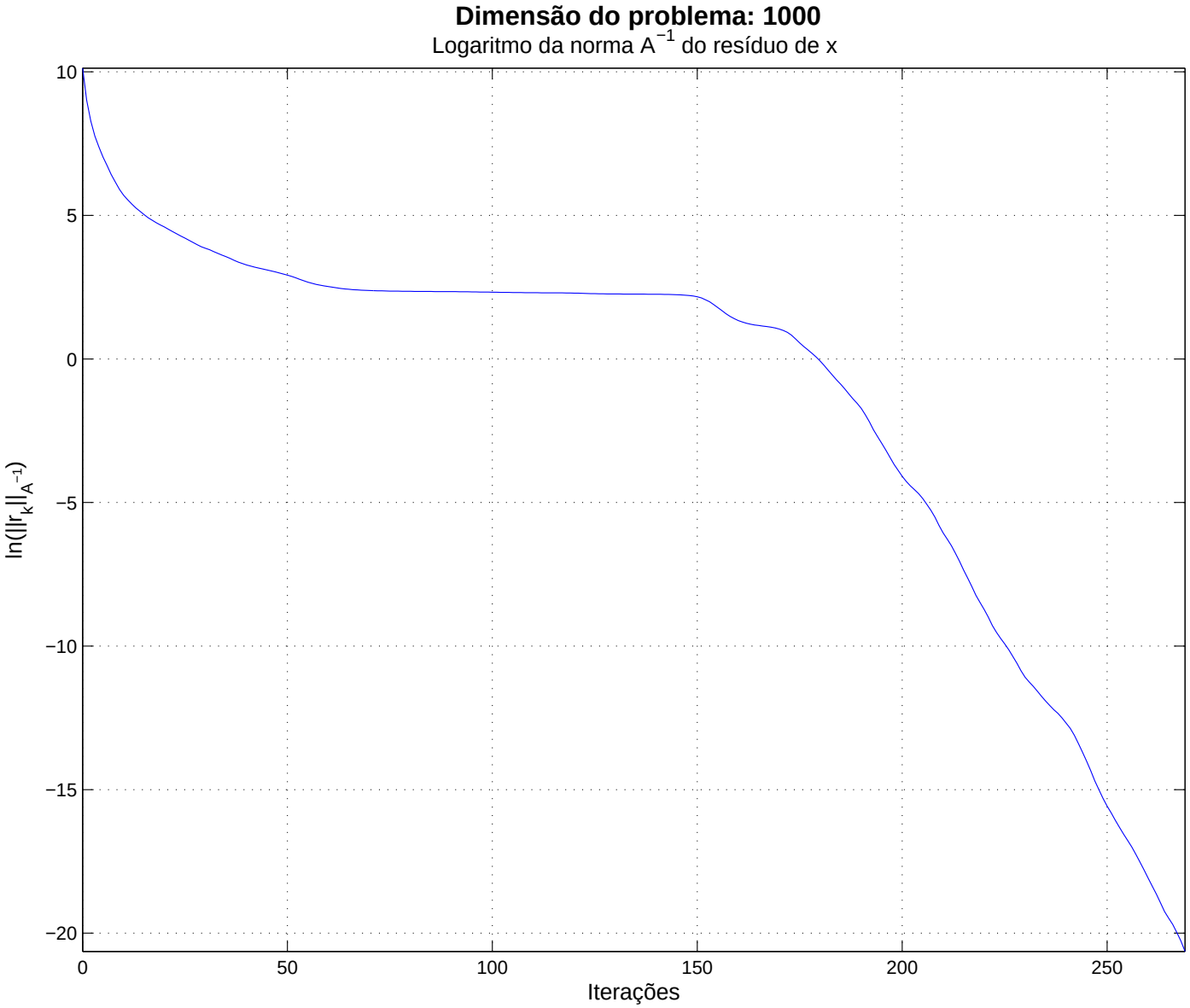

Figura 3.2: Convergência do CG em norma A^{-1} do resíduo

$$x_{k+1} = x_k + \alpha_k d_k \tag{3.27}$$

$$r_{k+1} = r_k + \alpha_k A d_k \tag{3.28}$$

$$d_{k+1} = r_{k+1} + \beta_k d_k \tag{3.29}$$

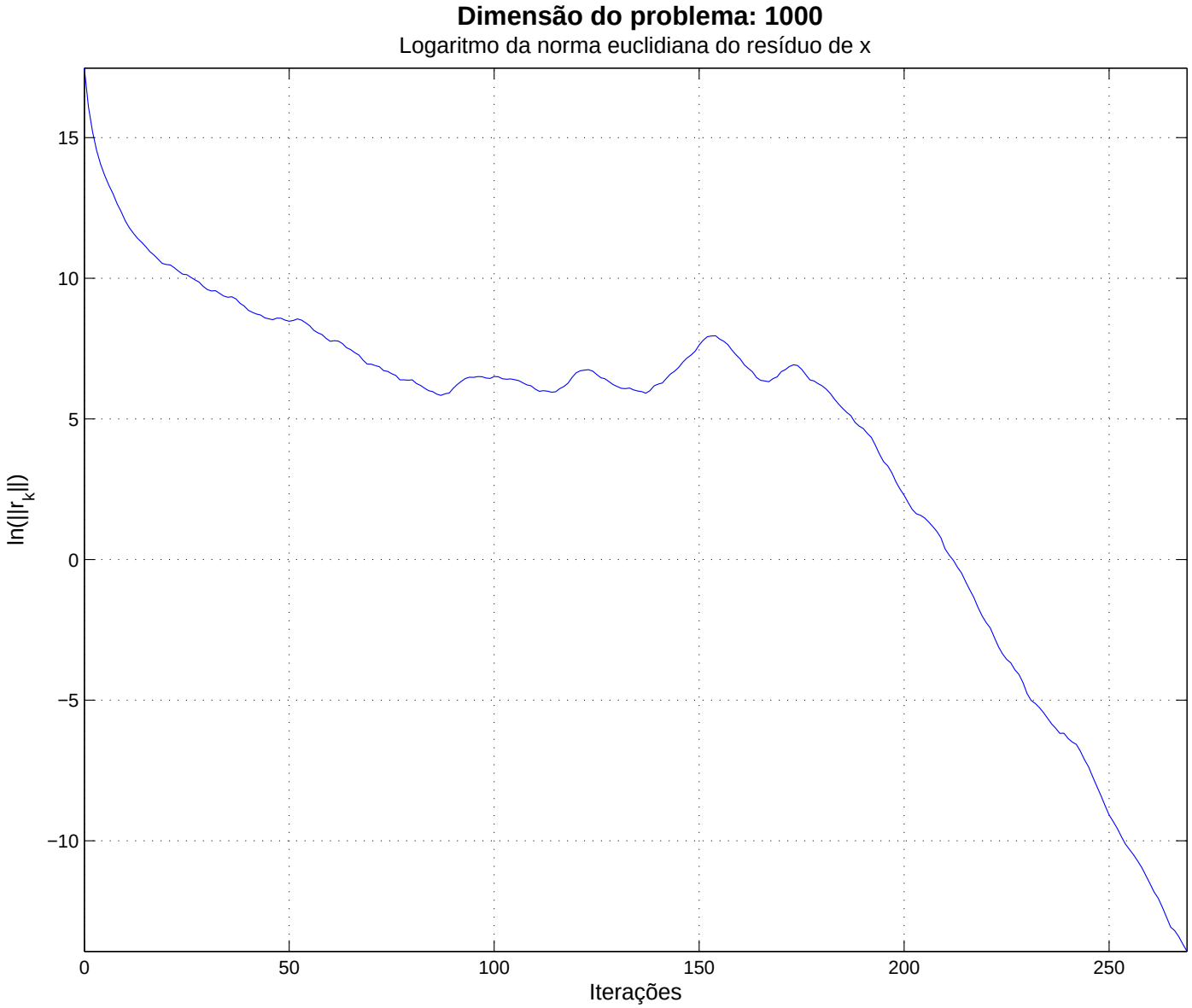

Figura 3.3: Convergência do CG em norma euclidiana do resíduo

que são as expressões do método CG. Supondo d_k e r_k como as variáveis de estado, podemos interpretar α_k e β_k como parâmetros de controle e dessa maneira, formarão um sistema bilinear. O objetivo do controle é conduzir d_k e r_k para zero, utilizando-se uma função de Lyapunov. Será utilizada a norma A^{-1} para (3.28) e a norma A para (3.29). São definidas as seguintes funções de Lyapunov:

$$V_r = \|r_k\|^2_{A^{-1}} \tag{3.30}$$

$$V_d = \|d_k\|^2_A \tag{3.31}$$

Calculando-se a variação da função V_r, conforme (3.32), temos:

$$\Delta V_r := r^T_{k+1} A^{-1} r_{k+1} - r^T_k A^{-1} r_k \tag{3.32}$$

$$\Delta V_r := -2\alpha_k r^T_k d_k + \alpha^2_k d^T_k A d_k \tag{3.33}$$

A escolha de α_k é aquela que minimiza ΔV_r, isto é, a que faz com que $\frac{\partial \Delta V_r}{\partial \alpha_k} = 0$, o que fornece exatamente o valor encontrado para α_k no método CG, conforme se vê a seguir.

$$\alpha_k = -\frac{d^T_k r_k}{d^T_k A d_k} \tag{3.34}$$

O que torna o valor de ΔV_r tão negativo quanto possível, conforme se vê em (3.35).

$$\Delta V_r = -\frac{(d^T_k r_k)^2}{d^T_k A d_k} \tag{3.35}$$

Conforme visto em [19], a derivação de α_k nos fornece uma pista em relação à robustez do método, uma vez que não foi utilizada nenhuma informação a respeito das propriedades dos vetores d_k (tal como A-ortogonalidade). Isto indica que, em implementações sob precisão finita, mesmo que as propriedades de A-ortogonalidade sejam perdidas, a escolha de α_k em (3.34) assegura que a norma A^{-1} de r vai decrescer.

Procedendo de forma análoga, isto é, calculando ΔV_d e derivando-se em relação a β_k para então se igualar a zero, obteremos o valor de β_k já mostrado para o CG, conforme vemos em (3.36).

$$\alpha_k = -\frac{r^T_{k+1} A d_k}{d^T_k A d_k} \tag{3.36}$$

o que resulta em:

$$\|d_{k+1}\|^2_A = \|r_{k+1}\|^2_A - \frac{(d^T_k A r_{k+1})^2}{d^T_k A d_k} \tag{3.37}$$

que nos mostra que

$$\|d_{k+1}\|^2_A < \|r_{k+1}\|^2_A \tag{3.38}$$

A partir de (3.35), concluímos que r decresce em qualquer norma e, utilizando-se (3.38), vemos que d decresce em norma A, mostrando a convergência do método por uma abordagem de controle. A figura 3.4, vista em [19], ilustra o método CG como um sistema de controle clássico, representado na forma de uma planta padrão $P = \{I, I, A, 0\}$ com um controlador dinâmico não estacionário $C = \{(\beta_k\ I - \alpha_k A), I, \alpha_k I, 0\}$ nas variáveis d_k, x_k. Esse controlador é do tipo Proporcional-Derivativo, porém com os respectivos ganhos variantes no tempo (como uma função de k).

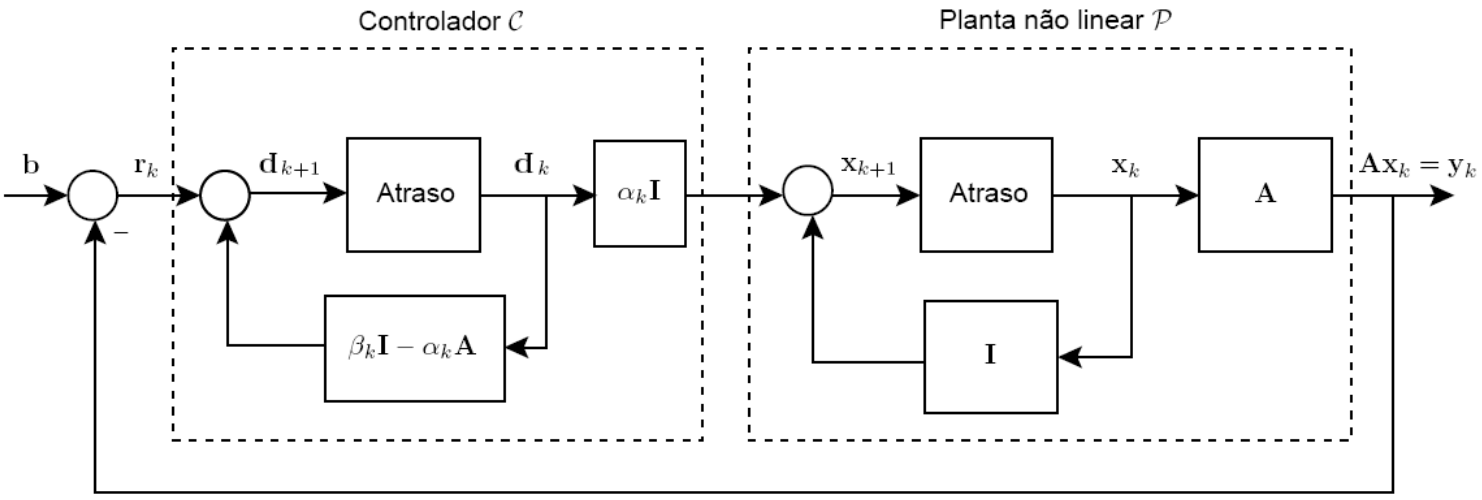

Figura 3.4: O método CG na forma de uma planta padrão

3.4 Uma forma paralela do CG

Uma outra abordagem do CG, que permitiria, a princípio, torná-lo paralelizável, pode ser vista em [13, p. 82]. A expressão de atualização dos resíduos e das direções pode ser vista a seguir.

$$r_{k+1} = r_k + \alpha_k d_k \tag{3.39}$$

$$d_{k+1} = r_k + \beta_k d_k \tag{3.40}$$

É interessante observar que as equações acima não utilizam a informação do resíduo r_{k+1}, diferentemente das equações do CG tradicional. Da mesma maneira, os passos α_k e β_k são calculados conforme se vê em (3.41) e (3.42), também sem utilizar informação sobre o resíduo r_{k+1}.

$$\alpha_k = \frac{r_k^T d_k}{d_k^T A d_k} \tag{3.41}$$

$$\beta_k = \frac{r_k^T A d_k}{d_k^T A d_k} \tag{3.42}$$

Dessa forma, o CG seria paralelizável, pois os cálculos de (3.39), (3.40), (3.41) e (3.42) poderiam ser feitos de maneira concorrente, já que os mesmos só utilizam as mesmas informações disponíveis desde o início da iteração, isto é, os vetores r_k e d_k.

No entanto, essa modificação para o CG apresenta resultados de desempenho significativamente pior, em comparação ao CG tradicional. Utilizando a mesma matriz 1000×1000 descrita nesta seção, com uma tolerância de 10^{-3}, o método proposto acima não converge, gerando denominadores nulos ou quase nulos nos cálculos de α_k e β_k. Essa piora de desempenho é mais um argumento para ilustrar que o CG, da maneira tradicional ou de maneiras similares, não pode ser paralelizável ou não apresenta bom desempenho nessas situações.

Capítulo 4

O método Gradiente Conjugado Cooperativo

Conforme visto no capítulo 3 deste trabalho, o método Gradiente Conjugado (CG) apresenta um excelente desempenho computacional no cálculo de soluções de sistemas lineares, desempenho esse que se mostra melhor que o de algoritmos com o mesmo propósito. Portanto, alguma melhora de desempenho com relação ao CG é um resultado interessante em termos computacionais.

Ultimamente, tem sido comum, em áreas como controle e computação distribuída, lançar mão da cooperação entre agentes a fim de atingir um objetivo comum. Em computação, tem se utilizado a paralelização, onde uma tarefa é dividida entre vários agentes (sendo cada agente correspondente a um núcleo de um processador). Como conseqüência dessa divisão, deve haver uma etapa de comunicação entre esses agentes e, dessa maneira, para minimizar o tempo de execução, os agentes devem operar da maneira mais simultânea possível, comunicando entre si somente quando necessário.

Em computadores antigos com vários processadores (em *chips* diferentes) a comunicação era um fator decisivo, pois a troca de informação entre os agentes se tornava lenta. Para agravar ainda mais esse fator, em alguns casos a memória de escrita desses processadores não era comum a todos, tornando o tempo gasto com as comunicações ainda maior. Outro problema decorrente da comunicação entre os processadores é a exigência do uso de barreiras para sincronismo, uma vez que em certas ocasiões um agente é forçado a esperar o outro (por exemplo quando um agente está escrevendo numa área de memória que um outro agente deseja ler). Esse sincronismo forçado também aumenta o tempo de execução total.

Ultimamente, com o desenvolvimento de memórias de maior capacidade, o problema do tempo gasto com a comunicação se tornou significativamente menor. Além disso, o advento de múltiplos núcleos de processamento num único *chip*, com acesso a uma

memória comum, tornou ainda mais eficiente a comunicação entre agentes.

A idéia que decorre naturalmente, tendo em vista esses avanços recentes, é a de paralelizar o algoritmo CG. No entanto, esse algoritmo não pode ser completamente paralelizado da maneira como foi proposto. Assim, um outro paradigma se faz necessário: o uso da cooperação.

A paralelização geralmente significa que cada um dos agentes é responsável por uma parcela do problema e, após o término da execução de cada agente, a resposta final é obtida pela resposta de cada um dos agentes. A cooperação, por outro lado, implica que todos os agentes calculem a solução de todo o problema, mas trocando informações entre si a respeito da solução. Ao final da execução, um dos agentes obtém a solução (sendo qualquer um dos agentes capaz de obtê-la), mas durante a execução houve troca de informação entre todos os agentes.

O conceito de troca de informação entre agentes foi introduzido em álgebra linear por Brezinski e Redivo-Zaglia em [6], antes mesmo do advento de processadores com múltiplos núcleos, conforme mencionado no capítulo 1 deste trabalho. Dessa maneira, o algoritmo CG é adaptado para uma versão cooperativa, nomeada **Gradiente Conjugado Cooperativo (cCG)**, a fim de permitir essa cooperação entre agentes. Utilizando-se P agentes, serão geradas P direções de busca, todas elas conjugadas em relação às direções anteriores. Quando os agentes (e resíduos) forem atualizados, eles o serão considerando essas P direções e, assim, P componentes do erro nas direções conjugadas serão anuladas por vez. Isso nos leva a entender que, dessa maneira, o algoritmo levará $\frac{N}{P}$ passos para anular todas as componentes do erro, desde que as direções produzidas a cada iteração sejam linearmente independentes entre si (se $\frac{N}{P}$ não for um número inteiro, o algoritmo anulará o erro em uma quantidade de passos igual ao número inteiro mais próximo de $\frac{N}{P}$, arredondado para cima).

Para que P componentes do erro sejam anuladas por iteração, é necessário que o passo α_k, que era originalmente um escalar, seja convertido em uma matriz $P \times P$. Isso porque cada um dos P agentes irá receber informação relativa a cada uma das P direções. Esse passo deverá garantir a minimização de todos os resíduos em norma A^{-1}, conforme visto no capítulo 3 deste trabalho. Da mesma maneira, para atualizar as direções o passo β_k deverá se transformar numa matriz $P \times P$, uma vez que cada uma das P direções receberá informação de todas as P direções anteriores. Este passo também deve ser escolhido de forma a garantir que cada uma das direções seja conjugada a todas as direções anteriores.

4.1 Construção do método

Será apresentado neste capítulo um exemplo de aplicação com 2 agentes (isto é, $P = 2$). Inicialmente, o esquema iterativo proposto em (3.2) é convertido no seguinte esquema, em que x_k e y_k são as soluções computadas por cada agente após o k-ésimo passo, $d_{x,k}$ e $d_{y,k}$ são as direções de busca para cada um dos agentes no k-ésimo passo e $\alpha_{ij,k}$ é o elemento da i-ésima linha e j-ésima coluna da matriz α_k. É suposto, por simplicidade que o valor de N (dimensão do problema) é par.

$$\begin{aligned} x_{k+1} &= x_k + \alpha_{11,k} d_{x,k} + \alpha_{12,k} d_{y,k} \\ y_{k+1} &= y_k + \alpha_{21,k} d_{x,k} + \alpha_{22,k} d_{y,k} \end{aligned} \tag{4.1}$$

Do ponto de vista do resíduo, o esquema iterativo de (4.1) se transforma no esquema observado a seguir. Os vetores $r_{x,k}$ e $r_{y,k}$ são os resíduos computados por cada agente no k-ésimo passo.

$$\begin{aligned} r_{x,k+1} &= r_{x,k} + \alpha_{11,k} A d_{x,k} + \alpha_{12,k} A d_{y,k} \\ r_{y,k+1} &= r_{y,k} + \alpha_{21,k} A d_{x,k} + \alpha_{22,k} A d_{y,k} \end{aligned} \tag{4.2}$$

A forma de se calcular a matriz α_k para o cCG segue a mesma filosofia de cálculo utilizada no CG, conforme foi visto no capítulo 3 deste trabalho, isto é, achar o valor de α_k que torne o resíduo na iteração seguinte mínimo em norma A^{-1}, derivando a expressão para o resíduo em relação ao passo e igualando a zero. A diferença no caso cooperativo é que temos dois resíduos e, para cada um deles, duas derivações parciais a se fazer (uma vez que há dois escalares e duas direções para cada resíduo). Esse procedimento, para os resíduos $r_{x,k+1}$ e $r_{y,k+1}$ pode ser visto a seguir.

$$\begin{bmatrix} \frac{\partial ||r_{x,k+1}||^2_{A^{-1}}}{\partial \alpha_{11,k}} \\ \frac{\partial ||r_{x,k+1}||^2_{A^{-1}}}{\partial \alpha_{12,k}} \end{bmatrix} = \begin{bmatrix} 2r_{x,k}^T d_{x,k} + 2\alpha_{11,k} d_{x,k}^T A d_{x,k} + 2\alpha_{12,k} d_{x,k}^T A d_{y,k} \\ 2r_{x,k}^T d_{y,k} + 2\alpha_{11,k} d_{x,k}^T A d_{y,k} + 2\alpha_{12,k} d_{y,k}^T A d_{y,k} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{4.3}$$

$$\begin{bmatrix} \frac{\partial ||r_{y,k+1}||^2_{A^{-1}}}{\partial \alpha_{21,k}} \\ \frac{\partial ||r_{y,k+1}||^2_{A^{-1}}}{\partial \alpha_{22,k}} \end{bmatrix} = \begin{bmatrix} 2r_{y,k}^T d_{x,k} + 2\alpha_{21,k} d_{x,k}^T A d_{x,k} + 2\alpha_{22,k} d_{x,k}^T A d_{y,k} \\ 2r_{y,k}^T d_{y,k} + 2\alpha_{21,k} d_{x,k}^T A d_{y,k} + 2\alpha_{22,k} d_{y,k}^T A d_{y,k} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{4.4}$$

Observando-se (4.3) e (4.4), podemos notar que há, em cada uma delas, um sistema

linear 2×2 a se resolver, sendo que cada um desses sistemas irá fornecer uma linha da matriz α_k. Os sistemas supracitados podem ser observados a seguir.

$$\begin{bmatrix} d_{x,k}^T A d_{x,k} & d_{x,k}^T A d_{y,k} \\ d_{x,k}^T A d_{y,k} & d_{y,k}^T A d_{y,k} \end{bmatrix} \begin{bmatrix} \alpha_{11,k} \\ \alpha_{12,k} \end{bmatrix} = - \begin{bmatrix} r_{x,k}^T d_{x,k} \\ r_{x,k}^T d_{y,k} \end{bmatrix} \tag{4.5}$$

$$\begin{bmatrix} d_{x,k}^T A d_{x,k} & d_{x,k}^T A d_{y,k} \\ d_{x,k}^T A d_{y,k} & d_{y,k}^T A d_{y,k} \end{bmatrix} \begin{bmatrix} \alpha_{21,k} \\ \alpha_{22,k} \end{bmatrix} = - \begin{bmatrix} r_{y,k}^T d_{x,k} \\ r_{y,k}^T d_{y,k} \end{bmatrix} \tag{4.6}$$

É possível perceber que, embora sejam dois sistemas a resolver, a matriz usada nos dois casos é a mesma, economizando esforço computacional por não ser necessário o cálculo da mesma duas vezes. Com os valores de α_k definidos, os vetores x_k e y_k são atualizados conforme mostrado em (4.1). Os resíduos são atualizados fazendo-se $r_{x,k+1} = Ax_{k+1} - b$ e $r_{y,k+1} = Ay_{k+1} - b$.

É interessante notar que os valores de $\alpha_{ij,k}$ obtidos pelos sistemas expostos em (4.5) e (4.6) são os valores que anulam a componente do erro nas direções $d_{x,k}$ e $d_{y,k}$. Tal fato pode ser observado exprimindo-se o erro inicial (de cada um dos agentes em função das $N/2$ direções geradas pelo método

$$\begin{aligned} e_{x,0} &= \sum_{j=0}^{\frac{N}{2}-1} \gamma_{x,j} d_{x,j} + \delta_{x,j} d_{y,j} \\ e_{y,0} &= \sum_{j=0}^{\frac{N}{2}-1} \gamma_{y,j} d_{x,j} + \delta_{y,j} d_{y,j} \end{aligned} \tag{4.7}$$

Pré-multiplicando-se as equações acima por $d_{x,k}^T A$ e $d_{y,k}^T A$, e lembrando que as direções de busca do passo atual são conjugadas às direções geradas em outros passos, isto é, $d_{x,k}^T A d_{x,j} = 0$, $d_{y,k}^T A d_{y,j} = 0$ e $d_{x,k}^T A d_{y,j} = 0, \forall k \neq j$.

$$d_{x,k}^T A e_{x,0} = \sum_{j=0}^{\frac{N}{2}-1} \gamma_{x,j} d_{x,k}^T A d_{x,j} + \delta_{x,j} d_{x,k}^T A d_{y,j} \tag{4.8a}$$

$$d_{x,k}^T A e_{x,0} = \gamma_{x,k} d_{x,k}^T A d_{x,k} + \delta_{x,k} d_{x,k}^T A d_{y,k} \tag{4.8b}$$

$$d_{x,k}^T A (e_{x,0} + \sum_{j=0}^{k-1} \alpha_{11,k} d_{x,k} + \alpha_{12,k} d_{y,k}) = \gamma_{x,k} d_{x,k}^T A d_{x,k} + \delta_{x,k} d_{x,k}^T A d_{y,k} \tag{4.8c}$$

$$d_{x,k}^T A e_{x,k} = \gamma_{x,k} d_{x,k}^T A d_{x,k} + \delta_{x,k} d_{x,k}^T A d_{y,k} \tag{4.8d}$$

$$d_{x,k}^T r_{x,k} = \gamma_{x,k} d_{x,k}^T A d_{x,k} + \delta_{x,k} d_{x,k}^T A d_{y,k} \tag{4.8e}$$

$$d_{y,k}^T A e_{x,0} = \sum_{j=0}^{\frac{N}{2}-1} \gamma_{x,j} d_{y,k}^T A d_{x,j} + \delta_{x,j} d_{y,k}^T A d_{y,j} \tag{4.9a}$$

$$d_{y,k}^T A e_{x,0} = \gamma_{x,k} d_{x,k}^T A d_{y,k} + \delta_{x,k} d_{y,k}^T A d_{y,k} \tag{4.9b}$$

$$d_{y,k}^T A (e_{x,0} + \sum_{j=0}^{k-1} \alpha_{11,k} d_{x,k} + \alpha_{12,k} d_{y,k}) = \gamma_{x,k} d_{x,k}^T A d_{y,k} + \delta_{x,k} d_{y,k}^T A d_{y,k} \tag{4.9c}$$

$$d_{y,k}^T A e_{x,k} = \gamma_{x,k} d_{x,k}^T A d_{y,k} + \delta_{x,k} d_{y,k}^T A d_{y,k} \tag{4.9d}$$

$$d_{y,k}^T r_{x,k} = \gamma_{x,k} d_{x,k}^T A d_{y,k} + \delta_{x,k} d_{y,k}^T A d_{y,k} \tag{4.9e}$$

$$d_{x,k}^T A e_{y,0} = \sum_{j=0}^{\frac{N}{2}-1} \gamma_{y,j} d_{x,k}^T A d_{x,j} + \delta_{y,j} d_{x,k}^T A d_{y,j} \tag{4.10a}$$

$$d_{x,k}^T A e_{y,0} = \gamma_{y,k} d_{x,k}^T A d_{x,k} + \delta_{y,k} d_{x,k}^T A d_{y,k} \tag{4.10b}$$

$$d_{x,k}^T A (e_{y,0} + \sum_{j=0}^{k-1} \alpha_{21,k} d_{x,k} + \alpha_{22,k} d_{y,k}) = \gamma_{y,k} d_{x,k}^T A d_{x,k} + \delta_{y,k} d_{x,k}^T A d_{y,k} \tag{4.10c}$$

$$d_{x,k}^T A e_{y,k} = \gamma_{y,k} d_{x,k}^T A d_{x,k} + \delta_{y,k} d_{x,k}^T A d_{y,k} \tag{4.10d}$$

$$d_{x,k}^T r_{y,k} = \gamma_{y,k} d_{x,k}^T A d_{x,k} + \delta_{y,k} d_{x,k}^T A d_{y,k} \tag{4.10e}$$

$$d_{y,k}^T A e_{y,0} = \sum_{j=0}^{\frac{N}{2}-1} \gamma_{y,j} d_{x,k}^T A d_{y,j} + \delta_{y,j} d_{y,k}^T A d_{y,j} \tag{4.11a}$$

$$d_{y,k}^T A e_{y,0} = \gamma_{y,k} d_{x,k}^T A d_{y,k} + \delta_{y,k} d_{y,k}^T A d_{y,k} \tag{4.11b}$$

$$d_{y,k}^T A (e_{y,0} + \sum_{j=0}^{k-1} \alpha_{21,k} d_{x,k} + \alpha_{22,k} d_{y,k}) = \gamma_{y,k} d_{x,k}^T A d_{y,k} + \delta_{y,k} d_{y,k}^T A d_{y,k} \tag{4.11c}$$

$$d_{y,k}^T A e_{y,k} = \gamma_{y,k} d_{x,k}^T A d_{y,k} + \delta_{y,k} d_{y,k}^T A d_{y,k} \tag{4.11d}$$

$$d_{y,k}^T r_{x,k} = \gamma_{x,k} d_{x,k}^T A d_{y,k} + \delta_{x,k} d_{y,k}^T A d_{y,k} \tag{4.11e}$$

Dessa maneira, podemos agrupar (4.8e), (4.9e), (4.10e) e (4.11e) em dois sistemas de equações lineares 2×2, conforme vemos em (4.12) e (4.13).

$$\begin{bmatrix} d_{x,k}^T A d_{x,k} & d_{x,k}^T A d_{y,k} \\ d_{x,k}^T A d_{y,k} & d_{y,k}^T A d_{y,k} \end{bmatrix} \begin{bmatrix} \gamma_{x,k} \\ \delta_{x,k} \end{bmatrix} = \begin{bmatrix} r_{x,k}^T d_{x,k} \\ r_{x,k}^T d_{y,k} \end{bmatrix} \tag{4.12}$$

$$\begin{bmatrix} d_{x,k}^T A d_{x,k} & d_{x,k}^T A d_{y,k} \\ d_{x,k}^T A d_{y,k} & d_{y,k}^T A d_{y,k} \end{bmatrix} \begin{bmatrix} \gamma_{y,k} \\ \delta_{y,k} \end{bmatrix} = \begin{bmatrix} r_{y,k}^T d_{x,k} \\ r_{y,k}^T d_{y,k} \end{bmatrix} \tag{4.13}$$

Vemos que os sistemas de (4.12) e (4.13) diferem dos sistemas (4.5) e (4.6) apenas pela ausência do sinal negativo no lado direito. Isso mostra que os valores das componentes dos erros em cada uma das direções conjugadas (γ's) e (δ's) são exatamente o oposto dos valores correspondentes de $\alpha_{ij,k}$. Portanto, os passos $\alpha_{ij,k}$ são exatamente os valores que zeram as componentes do erro nas direções conjugadas produzidas.

Uma vez que temos um método que zera duas componentes do erro em duas direções conjugadas por passo, devemos propor uma maneira de produzir duas direções que sejam conjugadas às duas direções anteriores. Para tal, devemos antes adaptar o esquema de atualização de direções de busca para o caso cooperativo, conforme vemos a seguir. O passo β_k, assim como o passo α_k, deve passar agora a ser uma matriz 2×2.

$$\begin{aligned} d_{x,k+1} &= r_{x,k+1} + \beta_{11,k} d_{x,k} + \beta_{12,k} d_{y,k} \\ d_{y,k+1} &= r_{y,k+1} + \beta_{21,k} d_{x,k} + \beta_{22,k} d_{y,k} \end{aligned} \tag{4.14}$$

Dado o esquema de atualização de direção de busca visto (4.14), deve-se fazer com que cada uma das direções de busca no passo $k+1$ sejam conjugadas em relação às duas direções de busca no passo k. Esse procedimento é visto a seguir.

$$d_{x,k}^T A d_{x,k+1} = d_{x,k}^T A (r_{x,k+1} + \beta_{11,k} d_{x,k} + \beta_{12,k} d_{y,k}) = 0 \tag{4.15a}$$

$$d_{y,k}^T A d_{x,k+1} = d_{y,k}^T A (r_{x,k+1} + \beta_{11,k} d_{x,k} + \beta_{12,k} d_{y,k}) = 0 \tag{4.15b}$$

$$d_{x,k}^T A d_{y,k+1} = d_{x,k}^T A (r_{y,k+1} + \beta_{21,k} d_{x,k} + \beta_{22,k} d_{y,k}) = 0 \tag{4.15c}$$

$$d_{y,k}^T A d_{y,k+1} = d_{y,k}^T A (r_{y,k+1} + \beta_{21,k} d_{x,k} + \beta_{22,k} d_{y,k}) = 0 \tag{4.15d}$$

Agrupando-se (4.15a) a (4.15d), podemos obter dois sistemas lineares 2×2 para o cálculo de β_k, quais sejam:

$$\begin{bmatrix} d_{x,k}^T A d_{x,k} & d_{x,k}^T A d_{y,k} \\ d_{x,k}^T A d_{y,k} & d_{y,k}^T A d_{y,k} \end{bmatrix} \begin{bmatrix} \beta_{11,k} \\ \beta_{12,k} \end{bmatrix} = - \begin{bmatrix} r_{x,k+1}^T A d_{x,k} \\ r_{x,k+1}^T A d_{y,k} \end{bmatrix} \tag{4.16}$$

$$\begin{bmatrix} d_{x,k}^T A d_{x,k} & d_{x,k}^T A d_{y,k} \\ d_{x,k}^T A d_{y,k} & d_{y,k}^T A d_{y,k} \end{bmatrix} \begin{bmatrix} \beta_{21,k} \\ \beta_{22,k} \end{bmatrix} = - \begin{bmatrix} r_{y,k+1}^T A d_{x,k} \\ r_{y,k+1}^T A d_{y,k} \end{bmatrix} \tag{4.17}$$

Podemos notar que a matriz dos dois sistemas que fornecem β_k é a mesma e, mais ainda, é a mesma dos sistemas que fornecem α_k. Isso, mais uma vez, economiza esforço computacional pois, para todos os sistemas resolvidos por iteração, a matriz é a mesma.

Uma vez definido o esquema de atualização das direções, o algoritmo está completo. Generalizando os cálculos supracitados para um valor de P qualquer ($P \geq 2$), podemos observar o pseudo-código do algoritmo Gradiente Conjugado Cooperativo (cCG) no algoritmo 4.1. As soluções de cada agente $(x_{1,k}, ..., x_{P,k})$, são concatenadas lado a lado na matriz X_k, na qual o vetor $x_{i,k}$ ocupa a i-ésima coluna de X_k. O mesmo ocorre com os resíduos, na matriz R_k e com as direções, na matriz D_k. Todas essas matrizes são de dimensão $N \times P$. O vetor 1_p é um vetor do $\mathbb{R}^P$ no qual todas as P componentes são iguais a 1.

Algoritmo 4.1 Algoritmo Gradiente Conjugado Cooperativo

choose $X_0 \in \mathbb{R}^{N \times P}$ ▷ As linhas de X_0 devem ser l.i.
$R_0 := AX_0 - b1_p^T$
$D_0 := R_0$
$k := 0$
while posto(D_k) == P **do**
    $\alpha_k := -R_k^T D_k (D_k^T A D_k)^{-1}$
    $X_{k+1} := X_k + D_k \alpha_k^T$
    $R_{k+1} := AX_{k+1} - b1_p^T$
    $\beta_k := -R_{k+1}^T A D_k (D_k^T A D_k)^{-1}$
    $D_{k+1} := R_{k+1} + D_k \beta_k^T$
    $k \leftarrow k + 1$
end while

É possível entender o algoritmo cCG como o caso multivariável do controlador visto em [19] e no capítulo 3, isto é, o caso em que os parâmetros de controle α_k e β_k são matrizes e as variáveis de estados agora são P resíduos e P direções. É importante ressaltar que as condições iniciais devem ser linearmente independentes, isto é, a matriz X_0 deve ter posto completo, embora a condição de perda de posto não tenha sido observada nas implementações práticas. Dessa forma, embora as condições iniciais possam ser

Tabela 4.1: Operações feitas pelo algoritmo Gradiente Conjugado Cooperativo

Operação feita pelo agente i	Resultado	Dimensão do resultado	Número de multiplicações escalares feitas por i
$Ad_{i,k}$	AD_k	$N \times P$	N^2
$d_{i,k}^T AD_k$	$D_k^T AD_k$	$P \times P$	NP
$r_{i,k}^T D_k$	$R_k^T D_k$	$P \times P$	NP
$\alpha_{i,k}(D_k^T AD_k) = r_{i,k}^T D_k$	$\alpha_k = R_k^T D_k (D_k^T AD_k)^{-1}$	$P \times P$	$\frac{P(P+1)(2P+1)}{6}$
$x_{i,k+1} = x_{k,i} + D_k \alpha_{k,i}^T$	$X_{k+1} = X_k + D_k \alpha_k^T$	$N \times P$	NP
$r_{i,k+1} = r_{i,k} + AD_k \alpha_{k,i}^T$	$R_{k+1} = R_k + AD_k \alpha_{k,i}^T$	$N \times P$	NP
$r_{i,k+1}^T AD_k$	$R_{k+1}^T AD_k$	$P \times P$	NP
$\beta_{k,i}(D_k^T AD_k) = -r_{i,k+1}^T AD_k$	$\beta_k = -R_{k+1}^T AD_k (D_k^T AD_k)^{-1}$	$P \times P$	$\frac{P(P+1)(2P+1)}{6}$
$d_{i,k+1} = r_{k+1,i} + D_k \beta_{k,i}^T$	$D_{k+1} = R_{k+1} + D_k \beta_k^T$	$N \times P$	NP

geradas aleatoriamente, é necessário que seja verificado se as mesmas formam um conjunto linearmente independente.

4.2 Complexidade do algoritmo

Podemos, mais uma vez, fazer um levantamento da quantidade de operações (multiplicações entre escalares) feitas pelo algoritmo. Dessa vez, no entanto, faremos um levantamento das operações feitas por cada um dos P agentes. Esse levantamento é observado na tabela 4.1.

Pela tabela 4.1 vemos que o algoritmo CG realiza ao todo, por iteração, a quantidade de multiplicações entre escalares vista em (4.18).

$$N^2 + 6NP + \frac{P(P+1)(2P+1)}{3} \tag{4.18}$$

Podemos ver que o algoritmo cCG também executa $O(N^2)$ operações por iteração e, como a quantidade de iterações necessárias para a convergência é no máximo $\frac{N}{P}$, a princípio poderia se afirmar que o algoritmo realiza $O(N^3)$ operações ao todo.

No entanto, a implementação do algoritmo 4.1 é teórica, pois considera aritmética exata. Considerando uma implementação em precisão finita, deve-se estabelecer um critério de parada, pois não será possível realizar cálculos exatos a cada iteração. Dessa maneira, o algoritmo cCG se torna um método iterativo. Uma outra nuance que deve ser observada é que na implementação com múltiplos agentes, há pontos nos quais deve haver um sincronismo, pois as informações de todos os agentes até aquele ponto devem estar disponíveis para todos os agentes dali em diante.

O primeiro desses pontos de sincronismo ocorre após o cálculo que cada agente faz de A vezes a sua direção $d_{i,k}$, pois no passo seguinte cada agente executa $d_{i,k}^T$ vezes a matriz

AD_k, que contém todos os produtos matriz direção. O segundo desses pontos ocorre após o cálculo de $d_{i,k}^T$ vezes a matriz AD_k por cada agente, pois a matriz $D_k^T AD_k$ deve ser completamente conhecida para os passos posteriores, onde ela é usada por cada um dos agentes no cálculo de cada uma das linhas de α_k e β_k (uma linha por agente).

Sendo assim, no algoritmo 4.2, vemos uma implementação prática do algoritmo cCG. Com essa implementação, o algoritmo cCG apresenta uma convergência rápida, seguindo os mesmos princípios do algoritmo CG tradicional, porém convergindo ainda mais rapidamente que o CG. Para ilustrar tal fato, é analisada a sua convergência novamente para a mesma matriz utilizada no capítulo 3, isto é, uma matriz com $N = 1000$ e número de condicionamento 10^6. O critério de parada é novamente a menor norma de resíduo ser menor do que 10^{-3}. Para exemplificar, usaremos $P = 2$.

Algoritmo 4.2 Algoritmo Gradiente Conjugado Cooperativo (Implementação prática)

choose $X_0 \in \mathbb{R}^{N \times P}$ ▷ As linhas de X_0 devem ser l.i.
$R_0 := AX_0 - b1_p^T$
$D_0 := R_0$
$k := 0$
minres := min($r_{i,0}$)
define tol ▷ tolerância para critério de parada
while minres $>$ tol **do**
    compute AD_k
    Barrier ▷ Ponto de sincronismo necessário
    compute $D_k^T AD_k$
    Barrier ▷ Ponto de sincronismo necessário
    $\alpha_k := -R_k^T D_k (D_k^T AD_k)^{-1}$
    $X_{k+1} := X_k + D_k \alpha_k^T$
    $R_{k+1} := AX_{k+1} - b1_p^T$
    $\beta_k := -R_{k+1}^T AD_k (D_k^T AD_k)^{-1}$
    $D_{k+1} := R_{k+1} + D_k \beta_k^T$
    $k \leftarrow k + 1$
    minres := min($r_{i,k+1}$)
end while

Podemos ver, na figura 4.1 a queda monotônica da norma A^{-1} dos dois resíduos, e na figura 4.2 a convergência oscilatória da norma euclidiana dos dois resíduos. Novamente, embora a queda da norma euclidiana não seja monotônica, o critério de parada é atingido de maneira rápida, em 169 iterações.

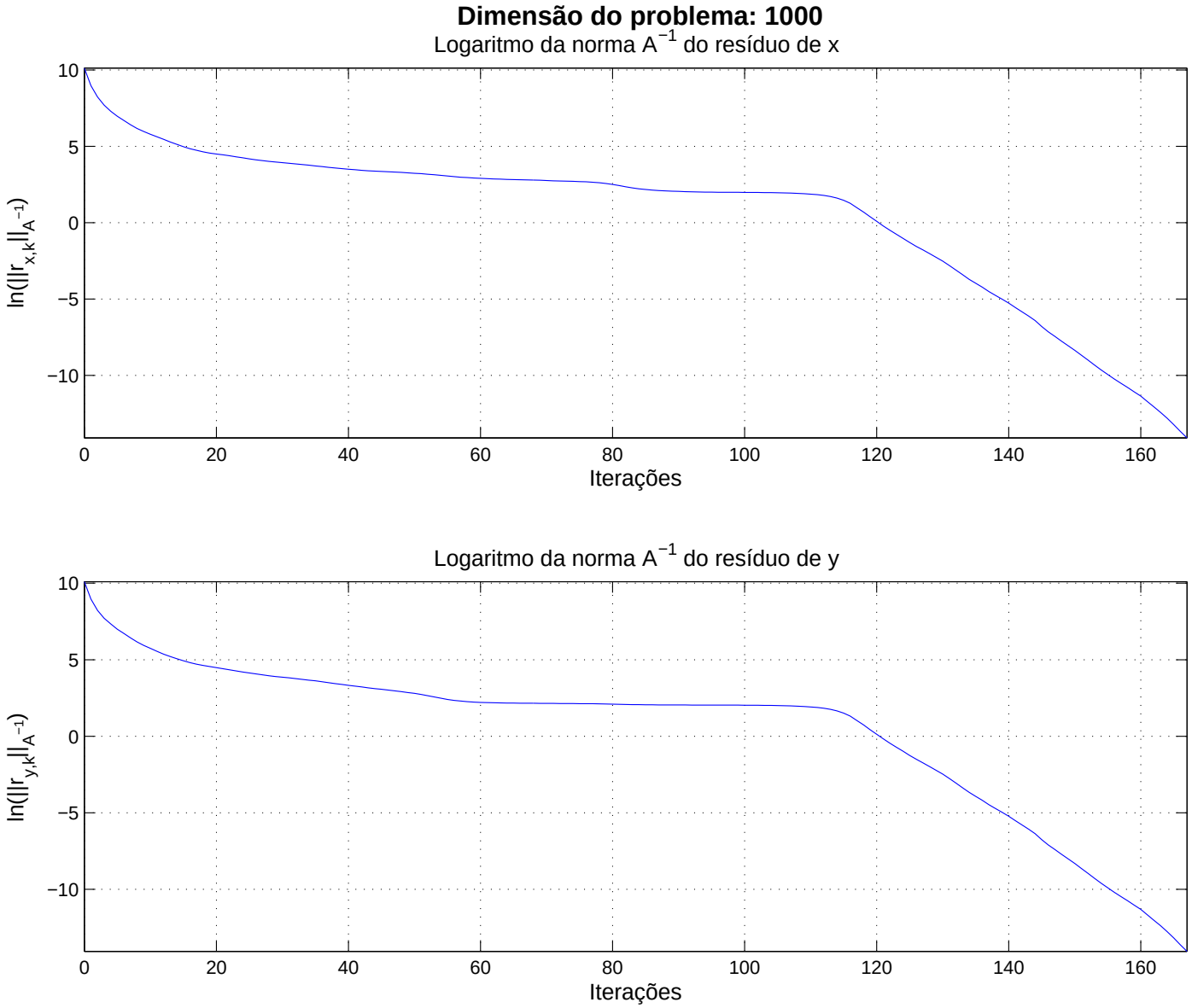

Figura 4.1: Convergência do cCG em norma A^{-1} do resíduo

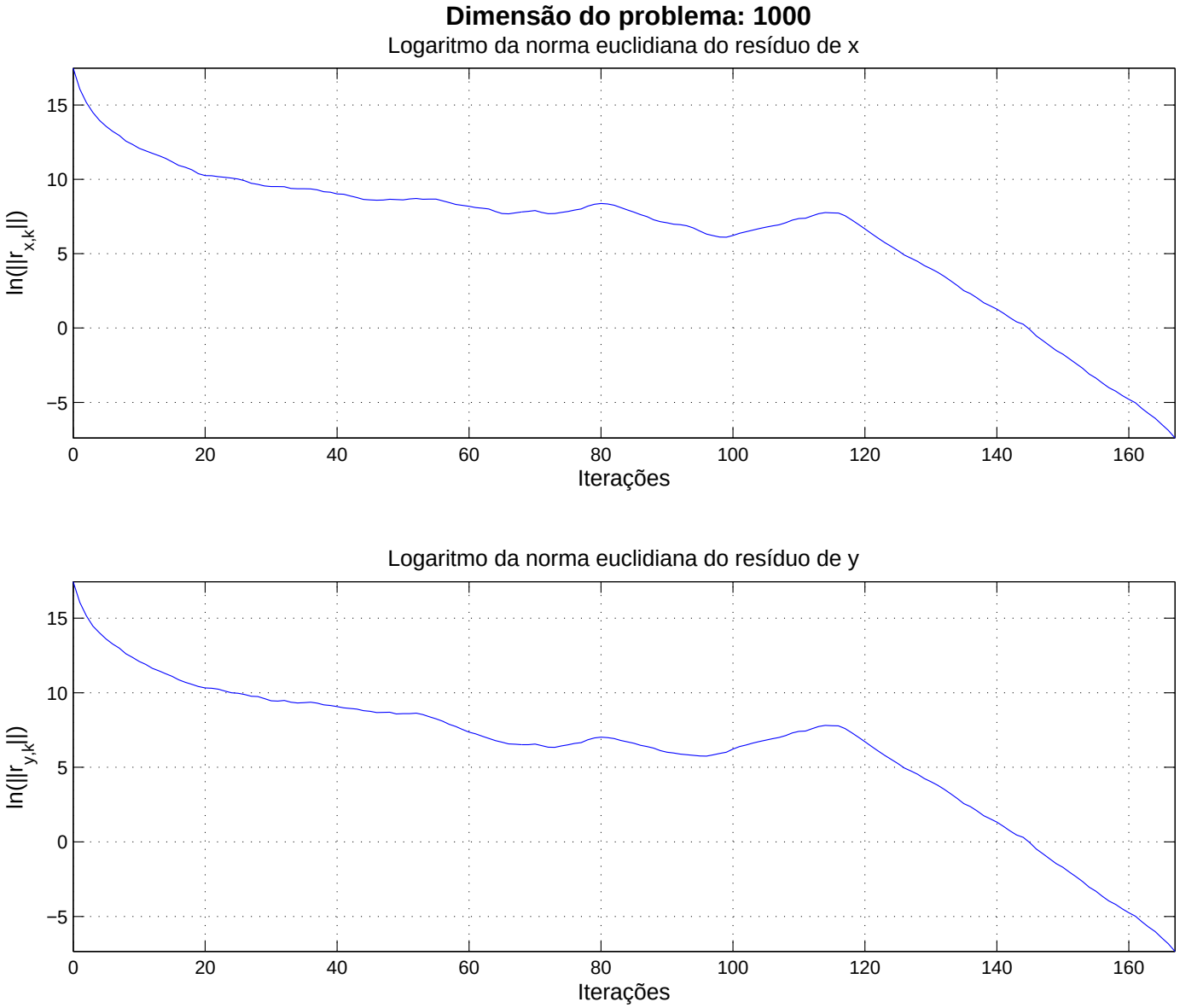

Figura 4.2: Convergência do cCG em norma euclidiana do resíduo

Já temos aqui um indício de que a convergência do cCG não só segue os mesmos princípios básicos do CG, mas ainda de que ela se dá de forma mais rápida. Os resultados experimentais detalhados a posteriori, na seção 6 deste trabalho, irão explorar melhor a comparação entre os algoritmos CG e cCG.

4.3 Derivações teóricas

O algoritmo cCG apresenta, assim como o CG, algumas propriedades interessantes. As mesmas podem ser observadas no teorema 4.1.

Teorema 4.1 (Propriedades do cCG)**.** *Enquanto a matriz D_k tiver posto completo (isto é, posto$(D_k) = P$)*

- *As matrizes $\{R_i\}_0^k$ são mutuamente ortogonais, as matrizes $\{D_i\}_0^k$ são mutuamente A-ortogonais, e os subespaços span$[R_i]_0^k$, span$[D_i]_0^k$ e span$[A^i R_0]_0^k$ são iguais e têm dimensão $(k+1)P$;*
- *Para qualquer e_j da base canônica do $\mathbb{R}^P$, o vetor $X_{k+1}e_j \in \mathbb{R}^N$ (que é formado pela j-ésima coluna de X_{k+1}, portanto igual ao vetor $x_{j,k+1}$) é o minimizador de $f(x_k) = \frac{x_k^T A x_k}{2} - b^T x$ no subespaço afim $X_0 e_j + span[D_i]_0^k$.*

O teorema 4.1 indica que, enquanto a matriz de resíduos R_k tiver posto completo, o algoritmo cCG se comporta essencialmente como o CG, fornecendo P estimativas diferentes na k-ésima iteração, sendo cada uma delas ótima em um subespaço afim construído a partir de uma das P condições iniciais e o espaço vetorial comum obtido a partir das colunas das matrizes-direção D_i, $i = 0, \ldots, k-1$. Esse espaço vetorial, $span[D_i]_0^k$, tem dimensão $(k+1)P$: cada iteração anula p direções. É importante ressaltar que diferentes colunas da matriz D_k (diferentes direções numa mesma iteração) não são necessariamente A-ortogonais (em outras palavras, $D_k^T A D_k$ não é necessariamente diagonal), mas, quando R_k tem posto completo, essas direções formam um conjunto de P vetores linearmente independentes. Essa afirmação, assim como a prova desse teorema são inspiradas pelas provas correspondentes para o CG tradicional dadas em [16, p. 270ff] e [18, p. 390-391].

Prova do teorema 4.1. • Primeiro mostramos que para qualquer k,

$$span\ [R_i]_0^k = span\ [D_i]_0^k\ . \tag{4.19}$$

• Mostramos o primeiro ponto por indução. Claramente, $D_j \perp_A D_k$ para qualquer $j < k$, e $span[R_i]_0^k = span[D_i]_0^k = span[A^i R_0]_0^k$ quando $k = 0$, não havendo nada a se verificar sobre as condições de ortogonalidade. Supondo-se que esta relação se verifica para algum k, e que se verifica para $i \leq k$, será mostrada a validade para $k+1$, supondo que R_{k+1} tem posto completo.

Das linhas 7 e 8 do algoritmo 4.1, $R_{k+1} = AX_{k+1} - 1_p^T b = R_k + AD_k \alpha_k^T$. Por indução, as colunas das matrizes R_k e AD_k estão no espaço $span[A^i R_0]_0^{k+1}$. Então,

$$R_{k+1} \in span\ [A^i R_0]_0^{k+1} \tag{4.20}$$

e conseqüentemente,

$$span\ [R_i]_0^{k+1} \subset span[A^i R_0]_0^{k+1} \tag{4.21}$$

Por outro lado, para qualquer vetor e_j da base canônica do $\mathbb{R}^P$,

$$R_{k+1}e_j \notin span\ [D_i]_0^k \tag{4.22}$$

porque cada resíduo é ortogonal as direções de busca anteriores, de forma que $R_{k+1}e_j \in span[D_i]_0^k$ para algum e_j implicaria em $R_{k+1}e_j = 0$, o que contradiz a afirmação de que R_{k+1} tem posto completo. De fato, pela mesma razão, também é possível mostrar que, para qualquer $v \in \mathbb{R}^P \setminus \{0\}$, $R_{k+1}v \notin span[D_i]_0^k$. Usando novamente o fato de que $span[R_{k+1}] = P$, é possível ver que

$$span\ [R_i]_0^{k+1} \supset span\ [A^i R_0]_0^{k+1} \tag{4.23}$$

e, de fato,

$$span\ [R_i]_0^{k+1} = span\ [A^i R_0]_0^{k+1}\ . \tag{4.24}$$

Similarmente, é possível mostrar pelas linhas 9 e 10 do algoritmo 4.1 que

$$span\ [D_i]_0^{k+1} \subset span\ [A^i R_0]_0^{k+1}\ , \tag{4.25}$$

e que a igualdade é obtida pelo mesmo argumento do posto.

A dimensão desses conjuntos é $(k+2)P$, pois eles contêm P vetores linearmente independentes no espaço $span[D_{k+1}]$ ortogonal a $span\ [D_i]_0^k$.

Da linha 10 do algoritmo 4.1, é possível mostrar que

$$D_i^T A D_{k+1} = D_i^T A(R_{k+1} + D_k \beta_k^T) \tag{4.26}$$

para $i < k$, o primeiro termo é zero porque $AD_i \in span[D_j]_0^{i+1}$ e os gradientes que formam as colunas de R_{k+1} são ortogonais a qualquer vetor em $span\ [D_j]_0^{i+1}$; o segundo termo também é zero pela hipótese da indução. Para $i = k$, o lado direito de (4.26) é zero porque β_k é escolhido precisamente para garantir essa propriedade. Assim, os vetores $\{D_i\}_0^{k+1}$ são mutuamente A-ortogonais.

A ortogonalidade de R_{k+1} vem das linhas 6 e 8 do algoritmo 4.1. A hipótese de indução foi provada para $k+1$, concluindo a prova da primeira parte do teorema 4.1.

• [Otimalidade]. Argumentando como o observado em [18, p.390-391], podemos escrever

$$X_{k+1}e_j = X_0 e_j + \sum_{i=0}^{k} D_i \gamma_i^T \tag{4.27}$$

para algum $\gamma_i \in \mathbb{R}^{1\times p}$. Otimalidade implica que

$$D_i^T(AX_{k+1}e_j - b) = 0, \tag{4.28}$$

Substituindo ((4.27)) em ((4.28)) e reescrevendo em termos das matrizes R_i e γ_i, vem

$$\gamma_i = -e_j^T R_0^T D_i (D_i^T A D_i)^{-1} \tag{4.29}$$

Por outro lado,

$$\nabla f(X_{k+1}e_j) = A(X_0 e_j + \sum_{i=0}^{k} D_i \gamma_i^T) - b = R_0 e_j + \sum_{i=0}^{k} A D_i \gamma_i^T \tag{4.30}$$

Então

$$D_{k+1}^T \nabla f(X_{k+1}e_j) = D_{k+1}^T R_0 e_j \tag{4.31}$$

Substituindo ((4.31)) em ((4.29)), nós obtemos:

$$\gamma_{k+1} = -\nabla f(X_{k+1}e_j)^T (D_{k+1}^T A D_{k+1})^{-1} = e_j^T \alpha_{k+1} \tag{4.32}$$

□

Uma questão natural é como analisar os casos nos quais, em algum ponto do algoritmo cCG, se obtém $posto(R_k) < P$. Na melhor das hipóteses, isso ocorre porque uma coluna de R_k é nula, por exemplo a j-ésima, significando que $\nabla f(X_k e_j) = 0$, e então que a k-ésima estimativa do j-ésimo agente é igual ao ótimo $x^* = A^{-1}b$. Mas, obviamente, $posto(R_k)$ pode ser menor que P sem qualquer coluna de R_k ser nula.

Primeiramente, o resultado do teorema 4.2 garante que essa perda de posto é, em geral, evitada durante a execução.

Teorema 4.2 (Genericidade da condição de posto completo da matriz de resíduos do cCG)**.** *Para um conjunto denso aberto de condições iniciais X_0 em $\mathbb{R}^{N\times P}$, se tem durante qualquer rodada do cCG*

$$\forall\ 0 \le k \le k^* \doteq \lfloor \frac{N}{P} \rfloor, posto(R_k) = P. \tag{4.33}$$

Além disso,

$$dimspan[D_i]_0^{k^*} = P\lfloor \frac{N}{P} \rfloor\ . \tag{4.34}$$

Em outras palavras: de forma geral, o algoritmo cCG pode ser rodado durante k^* passos, e

- *Quaisquer das colunas de X_{k^*} minimizam f no subespaço afim de $\mathbb{R}^N$ de codimensão $P\lfloor\frac{N}{P}\rfloor$;*
- *A aplicação do CG partindo de qualquer coluna de X_{k^*} leva a convergência em no máximo $N - P\lfloor\frac{N}{P}\rfloor \leq p - 1$ passos.*

A segunda parte do teorema 4.2 deve ser interpretada maneira a seguir. Quando a dimensão N da matriz A é um múltiplo do número P de agentes, o cCG termina em $\frac{n}{p}$ passos. Quando esse não é o caso, as estimativas X_{k^*} obtidas por $k = k^*$ minimizam a função f no subespaço afim cujo subespaço subjacente é $span[D_i]_0^{k^*}$ (ver 4.1). O interesse de (4.34) é mostrar que esse subespaço é bem grande: sua codimensão é $P\lfloor\frac{N}{P}\rfloor$, que é no máximo igual a $P - 1$.

Prova do teorema 4.2. O ponto principal consiste em mostrar que, genericamente, k^* iterações do algoritmo cCG podem ser feitas sem a perda de posto. Na realidade, os outros resultados da afirmação são conseqüências desse fato.

Para mostrar o último fato, é lançado mão o Teorema 4.1. Das propriedades listadas no mesmo, é possível ver que, para qualquer $0 \leq k \leq k^*$, o posto X_k não é completo se e somente se uma combinação linear dos P vetores coluna de X_0 pertence ao subespaço $span\ [D_i]_0^{k-1}$ de dimensão kP. Num espaço vetorial de dimensão $N > kp$, isso ocorre somente no complemento de um conjunto denso aberto.

Então, se os vetores coluna de X_k são linearmente independentes, o mesmo é verdade para R_k (ver a linha 8 do algoritmo 4.1, assim como para D_k, ver linha 10. Isso completa a prova do Teorema 4.2.

□

Utilizando-se (4.18), com as informações de operações realizadas por iteração e o fato de que o cCG converge em no máximo $\frac{N}{P}$ iterações, podemos escrever o teorema 4.3. A notação $\nu(P)$ será utilizada para definir quantas operações serão feitas ao todo pelo algoritmo cCG por cada agente, para uma dimensão N fixa.

Teorema 4.3 (Pior caso do tempo de execução multitarefa em aritmética exata)**.** *Genericamente, a execução multitarefa do cCG usando P agentes para um sistema linear como o de* (2.1) *de dimensão N necessita de*

$$\nu(P) = \frac{N^3}{P} + 6N^2 + N\frac{(P+1)(2P+1)}{3} \tag{4.35}$$

multiplicações feitas de forma síncrona em paralelo por cada processador, desde que a matriz D_k sempre mantenha o posto completo.

A fim de comprovar o teorema 4.3, e mostrar a A-ortogonalidade das direções produzidas, foram rodados os algoritmos CG e cCG com 2 agentes para uma matriz positiva definida de dimensão 10×10. Para uma matriz de dimensão pequena os erros numéricos serão desprezíveis, de forma que o CG deve produzir um resíduo de norma próxima a dimensão de máquina em 10 iterações, enquanto o cCG deve apresentar o mesmo comportamento em 5 iterações. A matriz utilizada é observada a seguir.

$$A = \begin{bmatrix} 1087.297 & 292.198 & -38.183 & 497.764 & -380.197 & -37.215 & 161.904 & 156.641 & -153.069 & -315.692 \\ 292.198 & 2536.747 & -53.709 & -314.636 & -77.127 & -104.663 & -403.671 & -403.810 & -163.634 & 441.556 \\ -38.183 & -53.709 & 3525.467 & 252.349 & 874.573 & 272.343 & 245.045 & 359.418 & 100.182 & -10.729 \\ 497.764 & -314.636 & 252.349 & 1811.358 & -754.439 & 381.576 & -325.066 & 902.820 & 171.056 & 289.251 \\ -380.197 & -77.127 & 874.573 & -754.439 & 1529.264 & -1049.319 & -352.537 & -143.696 & -571.914 & -33.059 \\ -37.215 & -104.663 & 272.343 & 381.576 & -1049.319 & 1786.647 & -293.313 & 306.400 & 531.420 & -614.497 \\ 161.904 & -403.671 & 245.045 & -325.066 & -352.537 & -293.313 & 3374.266 & -625.194 & 322.085 & -120.841 \\ 156.641 & -403.810 & 359.418 & 902.820 & -143.696 & 306.400 & -625.194 & 1375.962 & -387.662 & 97.088 \\ -153.069 & -163.634 & 100.182 & 171.056 & -571.914 & 531.420 & 322.085 & -387.662 & 2205.053 & 534.467 \\ -315.692 & 441.556 & -10.729 & 289.251 & -33.059 & -614.497 & -120.841 & 97.088 & 534.467 & 3611.473 \end{bmatrix} \tag{4.36}$$

Os resultados obtidos mostram que o CG produz um resíduo cuja norma é da ordem de 10^{-12} após exatas 10 iterações, enquanto o cCG produz um resíduo cuja norma é de ordem de grandeza aproximadamente igual (10^{-11}) em exatas 5 iterações, conforme esperado pelo teorema 4.3. A norma dos resíduos do CG e do cCG ao longo das iterações é observada nas tabelas 4.2 e 4.3, respectivamente.

Tabela 4.2: Convergência do CG para uma matriz de dimensão 10×10

Iteração	**Norma euclidiana do resíduo**
1	$6.846 \cdot 10^2$
2	$3.870 \cdot 10^2$
3	$2.170 \cdot 10^2$
4	$1.623 \cdot 10^2$
5	$4.958 \cdot 10^1$
6	$4.976 \cdot 10^1$
7	$8.278 \cdot 10^1$
8	$1.400 \cdot 10^1$
9	$1.281 \cdot 10^1$
10	$1.013 \cdot 10^{-12}$

Um outro resultado interessante, utilizando-se o Teorema 4.3 é o fato de que, para $N \geq 5$ sempre é vantajoso se utilizar $P \leq N$ processadores ao invés de um único. Esse resultado pode ser observado no Corolário 4.4.

Corolário 4.4 (Ganho *Multithread*)**.** *Para problemas de dimensão N maior ou igual a 5, é sempre vantajoso utilizar $P \leq N$ processadores ao invés de um único. Em outras palavras, quando $N \geq 5$,*

$$\forall P \in [1, N], \nu(1) \geq \nu(P). \tag{4.37}$$

Tabela 4.3: Convergência do cCG para uma matriz de dimensão 10×10

Iteração	**Norma euclidiana do resíduo do agente 1**	**Norma euclidiana do resíduo do agente 2**
1	$4.610 \cdot 10^2$	$9.003 \cdot 10^2$
2	$1.641 \cdot 10^2$	$3.144 \cdot 10^2$
3	$1.999 \cdot 10^2$	$1.605 \cdot 10^2$
4	$1.077 \cdot 10^2$	$1.377 \cdot 10^2$
5	$1.854 \cdot 10^{-11}$	$1.861 \cdot 10^{-11}$

Prova do Corolário 4.4.

$$\begin{aligned} \nu(1) - \nu(N) &= (N^3 + 2N) - (\frac{2N^3}{3} + 2N^2 + \frac{N}{3}) \\ \nu(1) - \nu(N) &= \frac{N}{3}(N^2 - 6N + 5) = \frac{N}{3}(N-1)(N-5) \end{aligned} \tag{4.38}$$

Além disso,

$$\frac{d\nu(1)}{dP} = N\left(\frac{7}{3} - N^2\right) \tag{4.39}$$

Que é negativo para $N \geq 2$, enquanto

$$\frac{d\nu(N)}{dP} = N\left(\frac{4N^2}{3}\right) \geq 0 \tag{4.40}$$

Uma vez que a derivada de ν é negativa em seu limite inferior ($\nu(1)$) e positiva em seu limite superior ($\nu(N)$), é provada a convexidade de ν. Essa convexidade leva, então, a conclusão de que $\nu(P) \leq \nu(1)$ para qualquer $1 \leq P \leq N$. □

Outro corolário interessante a se observar é o Corolário 4.5, que diz respeito a quantidade ótima de agentes a se utilizar e, mais ainda, a quantidade de operações realizada nessa condição de otimalidade, para $N \to +\infty$.

Corolário 4.5 (Ganho *multithread* ótimo)**.** *Para qualquer dimensão N do problema, existe um número ótimo P^* de processadores que minimiza $\nu(P)$. Além disso, quando $N \to +\infty$,*

$$P^* \approx \left(\frac{3}{4}\right)^{\frac{1}{3}} N^{\frac{2}{3}} \tag{4.41a}$$

O que leva a

$$\nu(P^*) \approx \left(\left(\frac{4}{3}\right)^{\frac{1}{3}} + \frac{2}{3}\left(\frac{3}{4}\right)^{\frac{2}{3}}\right) N^{2+\frac{1}{3}} \approx 1.651 N^{2+\frac{1}{3}} \tag{4.41b}$$

Prova do Corolário 4.5. A derivada de $\nu(P)$ é

$$\frac{d\nu(P)}{dP} = -\frac{N^3}{P^2} + \frac{4}{3}NP + N \tag{4.42}$$

Existe um único P^* que anula essa expressão. Esse valor atende a $N^2 = P^2(\frac{4}{3}P + 1)$, o que leva ao comportamento assintótico visto em (4.41a). Então, com o valor de P^*, o valor de (4.41b) é deduzido em seguida. □

O resultado do Corolário 4.5 é importante, pois mostra que o algoritmo cCG é capaz de resolver um sistema $Ax = b$ em $O(N^{2+\frac{1}{3}})$ operações por *thread*, o que pode ser comparado com resultados obtidos por eliminação Gaussiana ($O(N^3)$) ou outros resultados clássicos, vistos em [14].

Capítulo 5

Combinação Afim de Resíduos

No capítulo 4 deste trabalho, foi exposta uma maneira de converter o algoritmo do Gradiente Conjugado em uma versão cooperativa, fazendo com que múltiplos agentes resolvessem simultaneamente o problema, trocando informação através de suas direções de busca. Esse arranjo faz com que sejam geradas, por iteração, uma direção de busca para cada agente utilizado. Dessa maneira, ocorre a anulação de múltiplas componentes do erro a cada iteração.

No entanto, o conceito de cooperação exposto por Brezinski e Redivo-Zaglia em [6] não foi completamente explorado. Essa definição, conforme já foi visto, define a cooperação como "uma combinação de duas soluções aproximadas arbitrárias com coeficientes que somem 1". No algoritmo cCG exposto na seção 4 deste trabalho não há uma combinação entre as soluções computadas ou entre os resíduos; a cooperação se restringe as direções.

Para atingir esse objetivo, nesta seção é proposto uma etapa adicional ao algoritmo cCG, em que um dos agentes recebe informação dos demais através do resíduo e, conseqüentemente, da solução computada. Essa etapa pode se dar através de uma freqüência de ocorrência (ocorrendo a cada K iterações por exemplo, sendo um método determinístico) ou pode estar associada a uma probabilidade de ocorrência a cada iteração (sendo assim um método estocástico).

A proposta dessa etapa é ocorrer após a atualização dos resíduos e das soluções computadas, mas antes da atualização das direções. Em caso de ocorrência da mesma, deve-se escolher um resíduo para receber a comunicação afim dos demais resíduos (e dele mesmo). Supondo que esse resíduo seja o q-ésimo, teremos um esquema conforme se observa a seguir.

$$r_{q,k+1} = \sum_{i=1}^{P} \lambda_i r_{i,k+1}, \text{ onde } \sum_{i=1}^{P} \lambda_i = 1 \tag{5.1}$$

Sendo λ_i são os pesos atribuídos a cada resíduo. Uma vez que esses pesos somam 1

(uma combinação afim), é fácil mostrar que a mesma combinação se aplica as soluções computadas, conforme pode ser visto a seguir.

$$
\begin{aligned}
r_{q,k+1} &= \sum_{i=1}^{P} \lambda_i r_{i,k+1} \\
Ax_{q,k+1} - b &= \sum_{i=1}^{P} \lambda_i (Ax_{i,k+1} - b) \\
Ax_{q,k+1} - b &= \sum_{i=1}^{P} \lambda_i Ax_{i,k+1} - b \sum_{i=0}^{N-1} \lambda_i \\
Ax_{q,k+1} &= A \sum_{i=1}^{P} \lambda_i x_{i,k+1} \\
x_{q,k+1} &= \sum_{i=1}^{P} \lambda_i x_{i,k+1}
\end{aligned} \tag{5.2}
$$

Dessa maneira, deve-se calcular os valores de λ_i para, então, aplicá-los como pesos da combinação afim para as soluções computadas e para os resíduos. Para calcular tais pesos, deve-se utilizar um princípio similar ao utilizado para o cálculo do passo α_k, tanto no CG como no cCG. Isto é, deve-se minimizar a norma A^{-1} ao quadrado do resíduo que irá receber a comunicação afim. Para tal, devemos inicialmente avaliar essa norma considerando que todos os pesos λ_i somam 1. Isto implica que, na verdade, só podemos escolher $P-1$ pesos, devendo um deles ficar fixo. Aqui será escolhido o último peso (λ_P) como sendo fixo. Esse procedimento é observado a seguir.

$$
\begin{aligned}
r_{q,k+1} &= (\lambda_1 r_{1,k+1} + \lambda_2 r_{2,k+1} + \ldots + (1 - \lambda_1 - \lambda_2 - \ldots - \lambda_P) r_P) \\
r_{q,k+1} &= \lambda_1 \rho_1 + \lambda_2 \rho_2 + \ldots + \lambda_{P-1} \rho_{P-2} + r_{P,k+1}
\end{aligned} \tag{5.3}
$$

Sendo $\rho_i = r_{i,k+1} - r_{P,k+1}$, com i variando de 1 a $P-1$.

$$
||r_{q,k+1}||^2_{A^{-1}} = (\lambda_1 \rho_1 + \ldots \lambda_{P-1} \rho_{P-1} + r_{P,k+1})^T A^{-1} (\lambda_1 \rho_1 + \ldots \lambda_{P-1} \rho_{P-1} + r_{P,k+1}) \tag{5.4}
$$

A fim de minimizar a norma A^{-1} do resíduo exposta em (5.4), deve-se derivá-la com respeito a cada λ_i e então igualá-la a zero. Isso produzirá um sistema linear de equações, de ordem $(P-1) \times (P-1)$, conforme se vê em (5.5).

$$\begin{bmatrix} \rho_1^T A^{-1}\rho_1 & \rho_1^T A^{-1}\rho_2 & \dots & \rho_1^T A^{-1}\rho_{P-1} \\ \rho_1^T A^{-1}\rho_2 & \rho_2^T A^{-1}\rho_2 & \dots & \rho_2^T A^{-1}\rho_{P-1} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_1^T A^{-1}\rho_{P-1} & \rho_2^T A^{-1}\rho_{P-1} & \dots & \rho_{P-1}^T A^{-1}\rho_{P-1} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_{P-1} \end{bmatrix} = - \begin{bmatrix} r_{P,k+1}^T A^{-1}\rho_1 \\ r_{P,k+1}^T A^{-1}\rho_2 \\ \vdots \\ r_{P,k+1}^T A^{-1}\rho_{P-1} \end{bmatrix} \tag{5.5}$$

A princípio, é possível notar que para resolver o sistema visto em (5.5) é necessário conhecer a inversa de A, o que não é possível, conforme já foi visto. No entanto, ao expandir cada um dos termos $A^{-1}\rho_i$, obteremos a relação exposta em (5.6).

$$\begin{aligned} A^{-1}\rho_i = A^{-1}(r_{i,k+1} - r_{P,k+1}) = A^{-1}[(Ax_{i,k+1} - b) - (Ax_{P,k+1} - b)] = \\ = A^{-1}A(x_{i,k+1} - x_{P,k+1}) = (x_{i,k+1} - x_{P,k+1}) \end{aligned} \tag{5.6}$$

E, utilizando a relação de (5.6) e definindo-se $\sigma_i := x_{i,k+1} - x_{P,k+1}$, o sistema (5.5) se transforma no sistema a seguir.

$$\begin{bmatrix} \rho_1^T\sigma_1 & \dots & \rho_1^T\sigma_{P-1} \\ \rho_1^T\sigma_2 & \dots & \rho_2^T\sigma_{P-1} \\ \vdots & \ddots & \vdots \\ \rho_1^T\sigma_{P-1} & \dots & \rho_{P-1}^T\sigma_{P-1} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_{P-1} \end{bmatrix} = - \begin{bmatrix} r_{P,k+1}^T(x_{1,k+1} - x_{P,k+1}) \\ r_{P,k+1}^T(x_{2,k+1} - x_{P,k+1}) \\ \vdots \\ r_{P,k+1}^T(x_{P-1,k+1} - x_{P,k+1}) \end{bmatrix} \tag{5.7}$$

Dessa maneira, calcula-se os valores dos pesos de λ_1 a λ_{P-1}. O peso λ_P é calculado levando-se em conta que todos os pesos somam 1, isto é, fazendo-se 1 menos a soma de todos os outros pesos. O algoritmo Gradiente Conjugado Cooperativo considerando um passo de combinação afim dos resíduos é observado no algoritmo 5.1, já considerando uma implementação prática. São definidas as matrizes Δ_1, contendo os vetores $\rho_i = r_{i,k+1} - r_{P,k+1}$ empilhados lado a lado nas suas colunas (isto é, a i-ésima coluna de Δ_1 é igual a ρ_i) e Δ_2, contendo as diferenças entre os vetores $x_{i,k+1} - x_{P,k+1}$ (isto é, a i-ésima coluna de Δ_2 é igual a $x_{i,k+1} - x_{P,k+1}$). Ambas as matrizes são de dimensão $N \times P$.

A princípio, pode-se pensar que essa etapa não teria efeito, pois o cCG já é capaz de anular P componentes do erro por iteração. Se for realizada uma etapa intermediária que produz um resíduo (e um erro) menor, conforme vemos na figura 5.1, o erro produzido (e_1') poderia estar desalinhado em relação as direções conjugadas já encontradas, fato esse que será corrigido na iteração seguinte, conforme se vê na figura 5.2. Em outras palavras, embora seja produzido um resíduo menor pela comunicação afim, o próprio algoritmo cCG já seria capaz de minimizar o erro por si só, e esta etapa não faria diferença.

No entanto, devemos lembrar que, ao implementar o algoritmo num computador, não será disponível uma precisão aritmética exata. Neste cenário, o erro produzido pela

Algoritmo 5.1 Algoritmo Gradiente Conjugado Cooperativo com combinação afim

choose $X_0 \in \mathbb{R}^{N \times P}$ ▷ As linhas de X_0 devem ser l.i.
$R_0 := AX_0 - b1_p^T$
$D_0 := R_0$
$k := 0$
minres := min($r_{i,0}$)
define tol ▷ tolerância para critério de parada
while minres $>$ tol **do**
    compute AD_k
    Barrier ▷ Ponto de sincronismo necessário
    compute $D_k^T AD_k$
    Barrier ▷ Ponto de sincronismo necessário
    $\alpha_k := -R_k^T D_k (D_k^T AD_k)^{-1}$
    $X_{k+1} := X_k + D_k \alpha_k^T$
    $R_{k+1} := AX_{k+1} - b1_p^T$
    Barrier ▷ Ponto de sincronismo necessário para a combinação afim
    if Affine Combination **then**
        $\lambda := -(\Delta_1^T \Delta_2)^{-1}(r_{P,k+1}^T \Delta_2)$
        $\lambda_P := 1 - sum(\lambda)$
        $\lambda^T := [\ \lambda^T \quad \lambda_P\ ]$
        $r_q = R_{k+1}\lambda$ ▷ Supondo que o q-ésimo resíduo receba a combinação
    end if
    $\beta_k := -R_{k+1}^T AD_k (D_k^T AD_k)^{-1}$
    $D_{k+1} := R_{k+1} + D_k \beta_k^T$
    $k \leftarrow k + 1$
    minres := min($r_{i,k+1}$)
end while

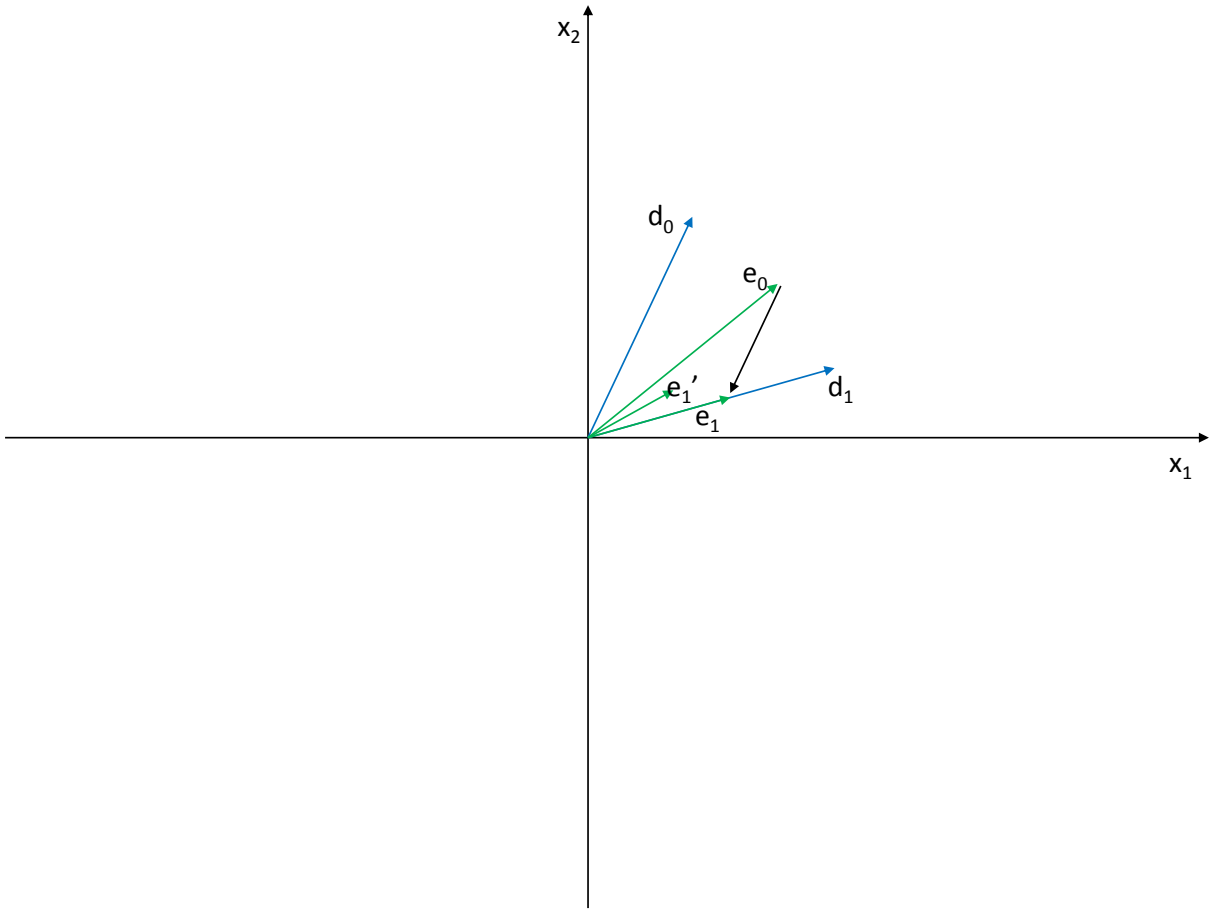

Figura 5.1: Erro produzido e'_1 pela combinação afim

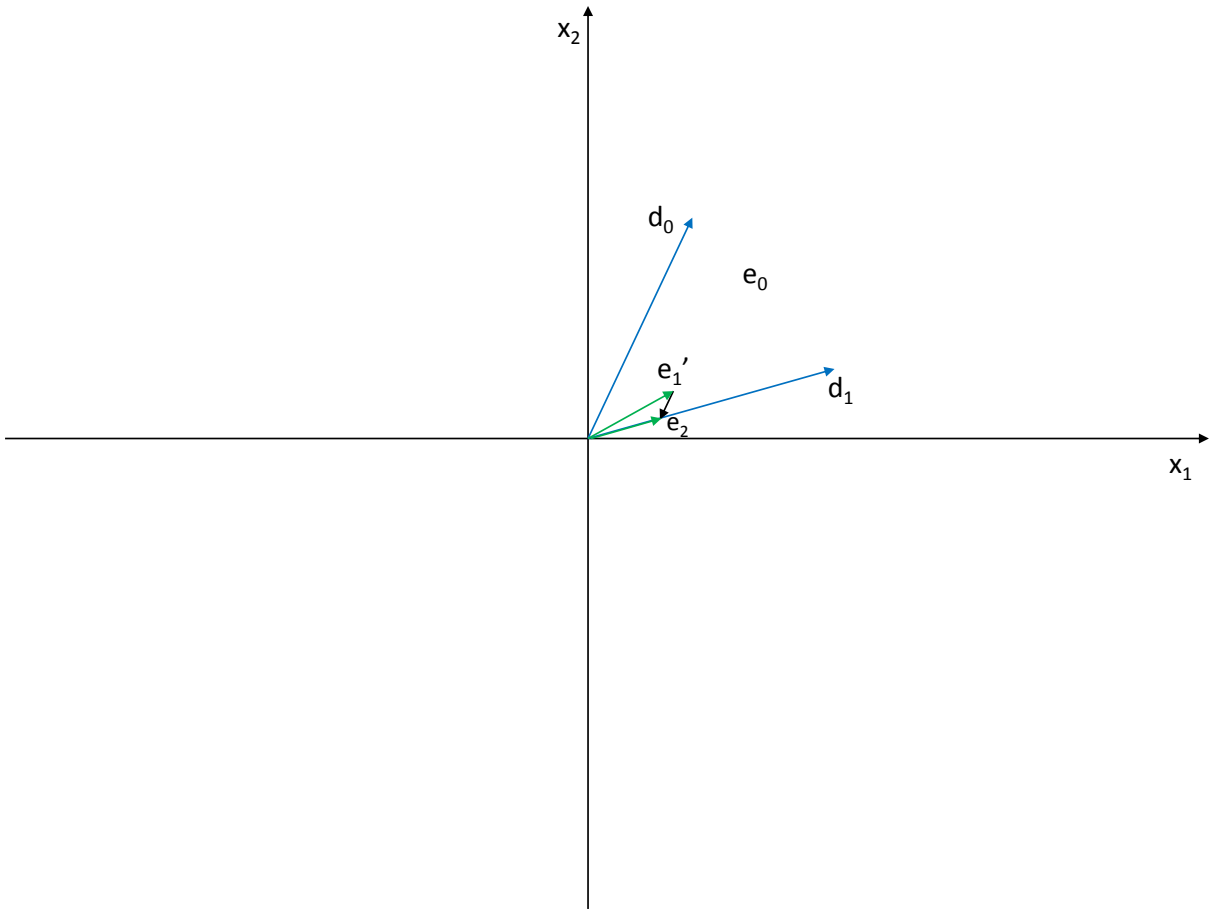

Figura 5.2: Erro e_2, corrigindo o erro produzido pela combinação afim

combinação afim pode apresentar uma norma tão pequena que, para a precisão disponível, a sua componente nas direções conjugadas geradas anteriormente continue nula, o que justificaria a sua utilização.

Isto é exatamente o que acontece e, para comprovar esse fato, é ilustrado novamente o exemplo da matriz utilizada nas seções 3 e 4 deste trabalho, com $N = 1000$ e condição 10^6. O critério de parada escolhido foi a norma do menor resíduo ser menor do que 10^{-3}. Foram utilizados dois agentes ($P = 2$), com a etapa de combinação afim ocorrendo a cada 10 iterações, de maneira alternada entre os dois agentes. Na figura 5.3 vemos a convergência em norma A^{-1} dos resíduos, enquanto na figura 5.4 é vista a convergência em norma euclidiana dos mesmos.

Mais uma vez, podemos observar que a convergência em norma A^{-1} é monotonicamente decrescente, embora a queda da norma euclidiana não seja monotônica. Mais importante, é possível notar que, na ocorrência de combinação afim, a norma A^{-1} do resíduo apresenta uma queda significativa. A combinação não afeta a norma euclidiana da mesma forma, fazendo-a crescer em alguns casos (embora também seja positiva na maioria das vezes), mas o desempenho do algoritmo com a combinação é melhor. Essa convergência se deu em 147 iterações, o que já ilustra uma vantagem de se usar esta etapa de combinação afim de resíduos.

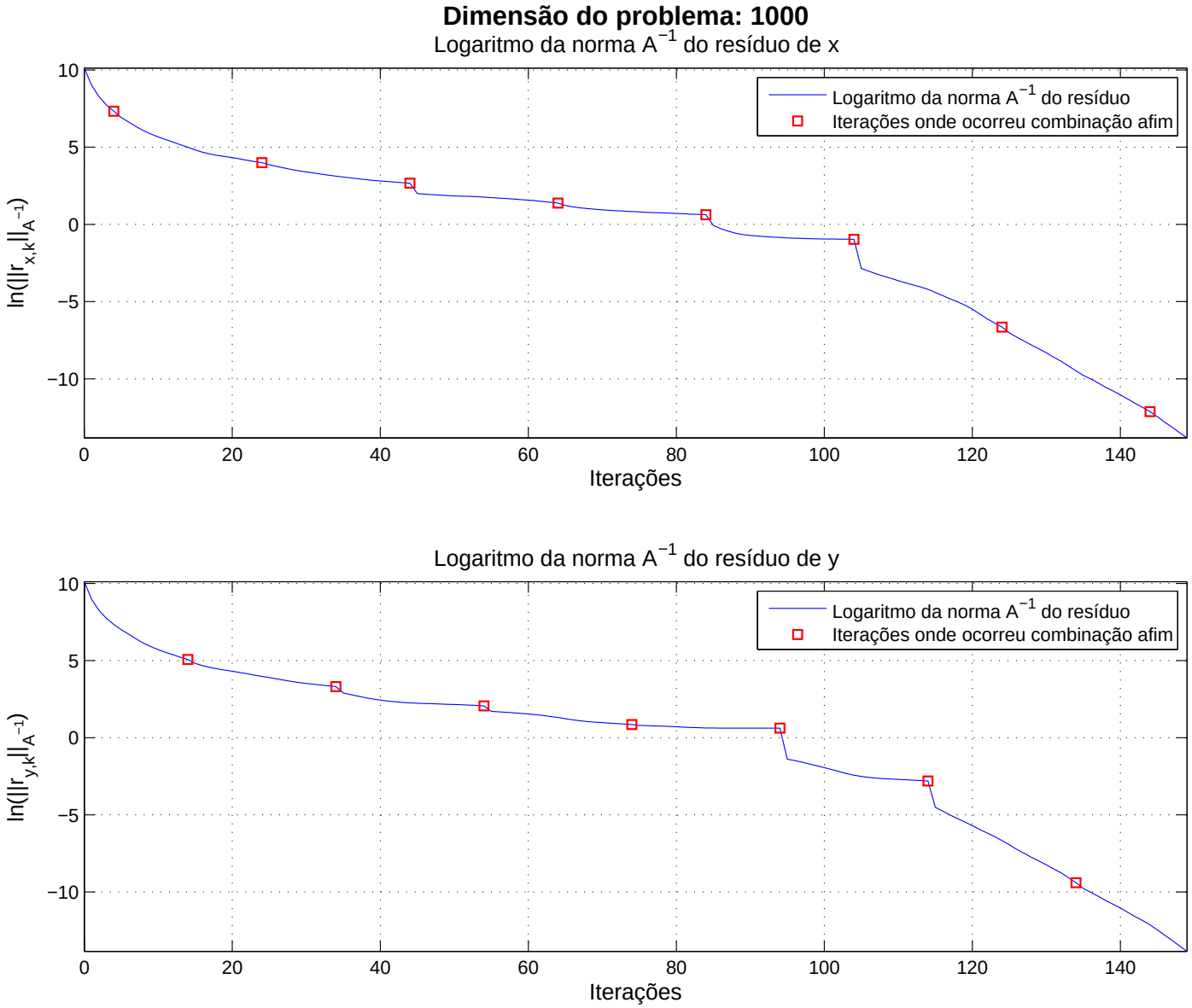

Figura 5.3: Convergência do cCG com combinação afim em norma A^{-1} do resíduo

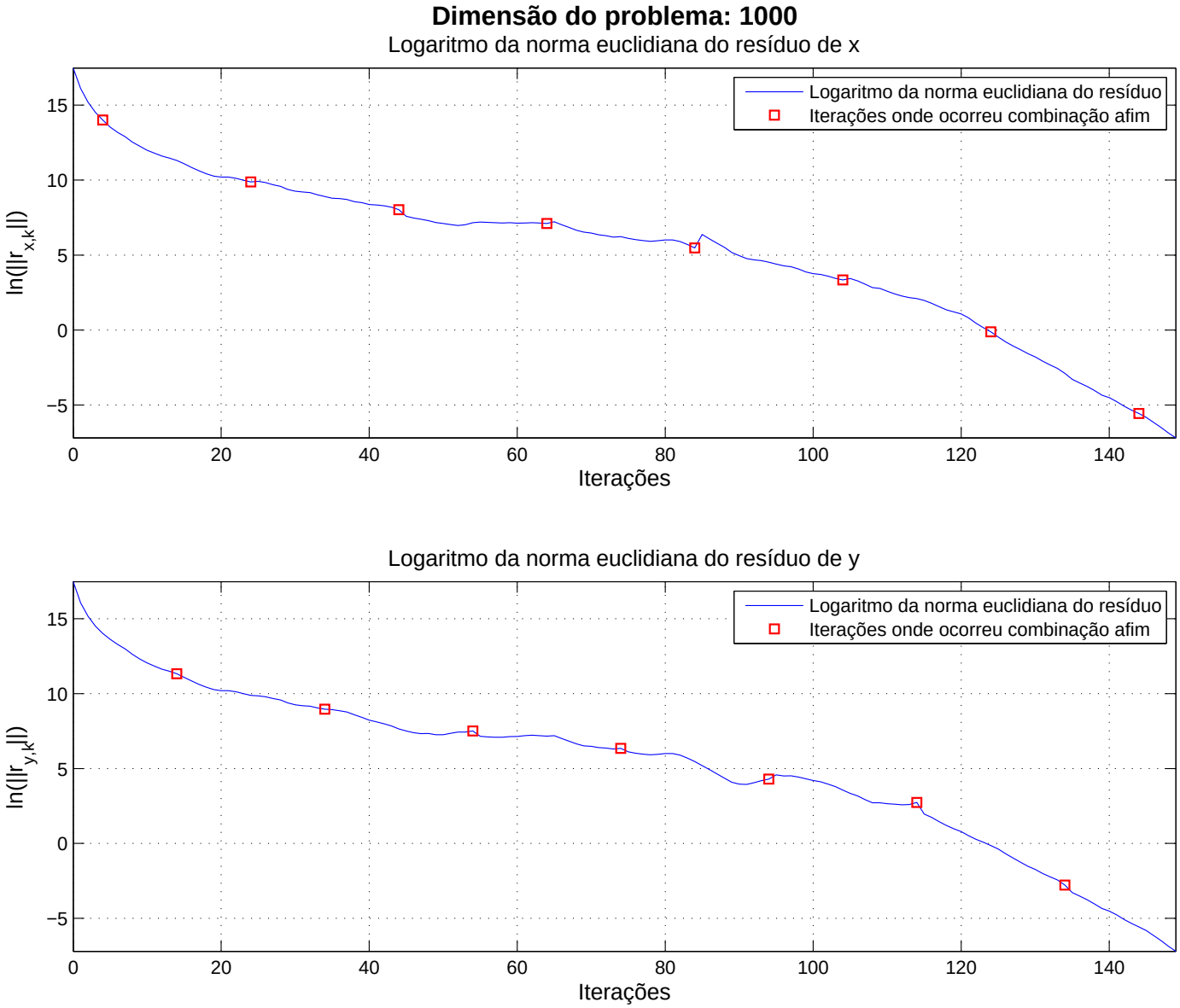

Figura 5.4: Convergência do cCG com combinação afim em norma euclidiana do resíduo

Capítulo 6

Resultados Experimentais

Uma vez que o algoritmo Gradiente Conjugado Cooperativo foi proposto e seu modelo de convergência foi levantado, faz-se mister a realização de testes para a validação desse modelo. Mas ainda, é necessário estabelecer uma comparação com o algoritmo Gradiente Conjugado clássico, a fim de estabelecer vantagens e desvantagens observando-se essa comparação.

Dessa maneira, inicialmente foi pensado em quantos agentes se deveria utilizar. Para tal, foi feito um levantamento prévio, utilizando inicialmente um código em C para gerar uma matriz positiva definida (detalhado no apêndice A.4 deste trabalho). De maneira simples, é escolhida a condição da matriz e então é gerada uma matriz diagonal Λ que apresenta essa condição. Em seguida, é gerada uma matriz ortogonal U pela ortogonalização de Gram-Schmidt, através de uma tradução para C do código de MATLAB proposto por Shilon em [20]. É escolhida a condição da matriz e então é escolhido o menor autovalor como sendo 1 e o maior autovalor sendo numericamente igual a condição. Os outros $N-2$ autovalores são gerados aleatoriamente por uma distribuição uniforme e, então, todos os N autovalores são armazenados numa matriz diagonal Λ. Por fim, é calculada a matriz $A = U\Lambda U^T$. Essa escolha de autovalores distribuídos uniformemente se justifica ao olharmos os teoremas 6.1 e 6.2, cujas provas se encontram em [12, pp. 299-300].

Teorema 6.1 (Convergência do algoritmo CG)**.**

- *Se A tem somente N autovalores distintos, então o algoritmo CG converge em no máximo N passos.*

Teorema 6.2 (Erro no algoritmo CG)**.**

- *Seja o CG aplicado ao problema $Ax = b$ com A positiva definida e com condição igual a κ. Então as normas-A dos erros satisfazem*

$$\frac{\|e_N\|_A}{\|e_0\|_A} \leq \frac{2}{\left[\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^N + \left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^{-N}\right]} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^N \tag{6.1}$$

Esse código foi compilado utilizando-se o compilador *GNU Compiler Collection* (GCC). Para os cálculos de álgebra linear foi utilizado o pacote *Linear Algebra Package* (LAPACK) e o pacote *Basic Linear Algebra Subprograms* (BLAS).

Uma vez gerada a matriz, foram escritos outros códigos em C. Um deles, detalhado no apêndice A.1, foi escrito para o algoritmo CG. O outro, detalhado no apêndice A.2, foi escrito para o algoritmo cCG (ainda sem a combinação afim, sendo o código para a introdução desse passo detalhado no apêndice A.3. Todos os códigos estão disponíveis para *download* em [21]. Os códigos foram compilados utilizando-se os mesmos pacotes e o mesmo compilador utilizados para gerar as matrizes PD, já citados. Para fazer a cooperação entre agentes, que necessita a inicialização de múltiplos *threads* (um em cada núcleo de processamento), foi utilizada a API *Open Multi Processing* (OMP). O processador utilizado foi um *Intel Core2Quad CPU Q8200* de 2.33 MHz. Esse processador apresenta 4 núcleos de processamento, permitindo, então, a inicialização de até 4 *threads* rodando em paralelo. Para este trabalho, foi escolhida a utilização de 3 agentes. A implementação final do cCG com três agentes pode ser observada no algoritmo 6.1.

É interessante notar que há uma pequena diferença na quantidade de operações feitas por agente em relação a tabela 4.1 da seção 4 deste trabalho. Uma vez que a matriz $D_k^T A D_k$, que pertence a $\mathbb{R}^{P\times P}$ é simétrica, não é necessário calcular todos P^2 os elementos dela, bastando calcular apenas $\frac{P(P+1)}{2}$ elementos. Dessa maneira, a quantidade de cálculos observada na tabela 6.1, para o caso com 3 agentes, é observada a seguir.

$$2N^2 + 14N + 28 \tag{6.2}$$

É interessante mencionar que é possível se programar o cCG fazendo-se N^2 operações por iteração, atualizando-se o resíduo da forma $r_{i,k+1} = r_{i,k} + AD_k\alpha_{k,i}^T$. Porém, a atualização do resíduo feita nesse trabalho é da forma exposta na tabela 6.1, o que consome $2N^2$ operações.

Em seguida, utilizando o cCG com 3 agentes, A seguir, foi avaliado o desempenho de matrizes encontradas no *website Matrix Market* [22]. Os algoritmos CG e cCG foram rodados para as matrizes **Bcsstk18**, de dimensão $N = 11948$ e condição igual a $6.5 \cdot 10^{11}$ e **fidap10**, de dimensão $N = 2410$ e condição $2.1 \cdot 10^{12}$. (ambas as matrizes são positivas definidas).

Para nenhuma dessas matrizes houve convergência, seja do algoritmo CG clássico, seja da versão cooperativa (cCG). É importante ressaltar que a condição dessas matrizes

Algoritmo 6.1 Implementação final do algoritmo cCG com 3 agentes

choose $x_0, y_0, z_0 \in \mathbb{R}^N$ ▷ Todos inicializados aleatoriamente com elementos entre -10 e 10. Os vetores devem ser l.i.
$r_{x,0} := Ax_0 - b$
$r_{y,0} := Ay_0 - b$
$r_{z,0} := Az_0 - b$
$d_{x,0} := r_{x,0}$
$d_{y,0} := r_{y,0}$
$d_{z,0} := r_{z,0}$
$k := 0$
$minres := min(norm(r_{x,0}), norm(r_{y,0}), norm(r_{z,0}))$
define tol ▷ Tolerância para critério de parada
while $minres > tol$ **do**
    ▷ Computar produtos entre A e todas as direções
    agent 1: compute $Ad_{x,k}$
    agent 2: compute $Ad_{y,k}$
    agent 3: compute $Ad_{z,k}$
    Barrier ▷ Sincronismo dos 3 agentes, para os próximos cálculos
    ▷ Cálculo de m_{ij}
    agent 1: $m_{11} := d_{x,k}^T Ad_{x,k}$; $m_{12} := d_{x,k}^T Ad_{y,k}$
    agent 2: $m_{13} := d_{x,k}^T Ad_{z,k}$; $m_{22} := d_{y,k}^T Ad_{y,k}$
    agent 3: $m_{23} := d_{y,k}^T Ad_{z,k}$; $m_{33} := d_{z,k}^T Ad_{z,k}$
    Barrier ▷ Sincronismo dos 3 agentes, para os próximos cálculos
    initialize $M := \{m_{ij}\}$ ▷ Matriz simétrica necessária para o cálculo de α e β, $m_{ij} = m_{ji}$
    ▷ Lados direitos necessários para o cálculo de α
    agent 1: $n_1 := \begin{bmatrix} r_{x,k}^T d_{x,k} & r_{x,k}^T d_{y,k} & r_{x,k}^T d_{z,k} \end{bmatrix}^T$
    agent 2: $n_2 := \begin{bmatrix} r_{y,k}^T d_{x,k} & r_{y,k}^T d_{y,k} & r_{y,k}^T d_{z,k} \end{bmatrix}^T$
    agent 3: $n_3 := \begin{bmatrix} r_{z,k}^T d_{x,k} & r_{z,k}^T d_{y,k} & r_{z,k}^T d_{z,k} \end{bmatrix}^T$
    ▷ Cálculo de α
    agent 1: Solve $M\alpha_1 = n_1$ ▷ Linha 1 de α
    agent 2: Solve $M\alpha_2 = n_2$ ▷ Linha 2 de α
    agent 3: Solve $M\alpha_3 = n_3$ ▷ Linha 3 de α
    ▷ Atualização de soluções computadas
    agent 1: $x_{k+1} := x_k + \alpha_{11} d_{x,k} + \alpha_{12} d_{y,k} + \alpha_{13} d_{z,k}$
    agent 2: $y_{k+1} := y_k + \alpha_{21} d_{x,k} + \alpha_{22} d_{y,k} + \alpha_{23} d_{z,k}$
    agent 3: $z_{k+1} := z_k + \alpha_{31} d_{x,k} + \alpha_{32} d_{y,k} + \alpha_{33} d_{z,k}$
    ▷ Atualização de resíduos
    agent 1: $r_{x,k+1} := Ax_{k+1} - b$
    agent 2: $r_{y,k+1} := Ay_{k+1} - b$
    agent 3: $r_{z,k+1} := Az_{k+1} - b$
    ▷ Lados direitos necessários para o cálculo de β
    agent 1: $n_1 := \begin{bmatrix} r_{x,k+1}^T Ad_{x,k} & r_{x,k+1}^T Ad_{y,k} & r_{x,k+1}^T Ad_{z,k} \end{bmatrix}^T$
    agent 2: $n_2 := \begin{bmatrix} r_{y,k+1}^T Ad_{x,k} & r_{y,k+1}^T Ad_{y,k} & r_{y,k+1}^T Ad_{z,k} \end{bmatrix}^T$
    agent 3: $n_3 := \begin{bmatrix} r_{z,k+1}^T Ad_{x,k} & r_{z,k+1}^T Ad_{y,k} & r_{z,k+1}^T Ad_{z,k} \end{bmatrix}^T$
    ▷ Cálculo de β
    agent 1: Solve $M\beta_1 = n_1$ ▷ Linha 1 de β
    agent 2: Solve $M\beta_2 = n_2$ ▷ Linha 2 de β
    agent 3: Solve $M\beta_3 = n_3$ ▷ Linha 3 de β
    ▷ Atualização das direções
    agent 1: $d_{x,k+1} := r_{x,k+1} + \beta_{11} d_{x,k} + \beta_{12} d_{y,k} + \beta_{13} d_{z,k}$
    agent 2: $d_{y,k+1} := r_{y,k+1} + \beta_{21} d_{x,k} + \beta_{22} d_{y,k} + \beta_{23} d_{z,k}$
    agent 3: $d_{z,k+1} := r_{z,k+1} + \beta_{31} d_{x,k} + \beta_{32} d_{y,k} + \beta_{33} d_{z,k}$
    ▷ Cálculo das normas
    agent 1: $norm_{r_x} = norm(r_{x,k+1})$
    agent 2: $norm_{r_y} = norm(r_{y,k+1})$
    agent 3: $norm_{r_z} = norm(r_{z,k+1})$
    $minres := min(norm_{r_x}, norm_{r_y}, norm_{r_z})$
    $k \leftarrow k + 1$
end while

Tabela 6.1: Operações feitas pelo algoritmo cCG com 3 agentes

Operação feita pelo agente i	Resultado	Dimensão do resultado	Número de multiplicações escalares feitas por i
$Ad_{i,k}$	AD_k	$N \times P$	N^2
$d_{i,k}^T AD_k$	$D_k^T AD_k$	$P \times P$	NP
$r_{i,k}^T D_k$	$R_k^T D_k$	$P \times P$	NP
$\alpha_{i,k}(D_k^T AD_k) = r_{i,k}^T D_k$	$\alpha_k = R_k^T D_k (D_k^T AD_k)^{-1}$	$P \times P$	$\frac{P(P+1)(2P+1)}{6}$
$x_{i,k+1} = x_{k,i} + D_k \alpha_{k,i}^T$	$X_{k+1} = X_k + D_k \alpha_k^T$	$N \times P$	NP
$r_{i,k+1} = Ax_{k,i} - b$	$R_{k+1} = AX_k - b1_p^T$	$N \times P$	N^2
$r_{i,k+1}^T AD_k$	$R_{k+1}^T AD_k$	$P \times P$	NP
$\beta_{k,i}(D_k^T AD_k) = -r_{i,k+1}^T AD_k$	$\beta_k = -R_{k+1}^T AD_k (D_k^T AD_k)^{-1}$	$P \times P$	$\frac{P(P+1)(2P+1)}{6}$
$d_{i,k+1} = r_{k+1,i} + D_k \beta_{k,i}^T$	$D_{k+1} = R_{k+1} + D_k \beta_k^T$	$N \times P$	NP
Total de multiplicações entre escalares por agente por iteração			$2N^2 + 5NP + \frac{P(P+1)(2P+1)}{3}$

é elevada, sendo portanto mais difícil a convergência. O efeito da condição da matriz sobre métodos iterativos é um fato amplamente conhecido e, dessa maneira, utilizar matrizes com condição elevada para estes métodos não gera bons resultados. No entanto, observando o resíduo iteração após iteração, o CG estaria mais próximo de convergir que o cCG.

Assim, para os testes que se seguiram, foram utilizadas matrizes cheias (ao contrário das matrizes esparsas do *Matrix Market*) geradas a partir de uma condição pré-estabelecida, através do código em linguagem C que pode se observar no apêndice A.4 deste trabalho. Essas matrizes foram rodadas nos algoritmos CG clássico (detalhado no apêndice A.1) e CG cooperativo (detalhado no apêndice A.2). Ainda foi considerada a etapa de combinação afim, cujo código é descrito no apêndice A.3.

A implementação da etapa de combinação afim com 3 agentes pode ser lida no algoritmo 6.2, e deve ser inserida entre as linhas 38 e 39 do algoritmo 6.1.

Algoritmo 6.2 Implementação da combinação afim com 3 agentes

```
1:  Barrier                                   ▷ Ponto de Sincronismo necessário para a combinação afim
2:  if Affine Combination then
3:      c_11 := (r_{x,k+1} − r_{z,k+1})^T (x_{k+1} − z_{k+1})
4:      c_12 := (r_{x,k+1} − r_{z,k+1})^T (y_{k+1} − z_{k+1})
5:      c_22 := (r_{y,k+1} − r_{z,k+1})^T (y_{k+1} − z_{k+1})
6:      initialize C := {c_ij}                          ▷ Matriz simétrica para o cálculo de λ, c_ij = c_ji
7:      n := [ −r_{z,k+1}^T (x_{k+1} − z_{k+1})   −r_{z,k+1}^T (y_{k+1} − z_{k+1}) ]   ▷ Lado direito para o cálculo de λ
8:      solve Cλ = n
9:      λ_1 := λ(1)
10:     λ_2 := λ(2)
11:     λ_3 := 1 − λ(1) − λ(2)
12:     choose q ∈ {1, 2, 3}
13:     r_{q,k+1} = λ_1 r_{x,k+1} + λ_2 r_{y,k+1} + λ_3 r_{z,k+1}
14: end if
```

Foram geradas várias matrizes, com a dimensão N variando de 1000 a 25000. Para

as matrizes de dimensão menor (até $N = 5000$) foram rodadas 20 condições iniciais diferentes. Para as maiores dimensões foram rodadas 10 condições iniciais, devido ao maior tempo de convergência para tais matrizes. Os dados médios de iterações gastas para convergência e tempo dos algoritmos CG, cCG sem combinação afim e cCG com combinação afim podem ser lidos na tabela 6.2. O acrônimo **acCG** é utilizado para se referir ao cCG com combinação afim.

Tabela 6.2: Dados de convergência para os algoritmos CG, cCG e cCG afim.

Dimensão da Matriz	**Número de iterações**			**Tempo (s)**		
	CG	**cCG**	**acCG**	**CG**	**cCG**	**acCG**
1000	245.50	137.05	131.85	1.80	1.05	1.00
1500	230.15	146.65	151.90	3.85	3.05	3.05
2000	303.35	182.60	171.30	8.05	5.45	4.95
2500	231.60	158.20	131.15	11.05	7.25	5.95
3000	347.00	221.35	219.15	20.65	15.30	14.90
3500	402.15	240.50	230.75	32.15	23.35	21.70
4000	399.85	257.45	238.90	40.30	31.85	30.70
4500	391.85	237.10	228.60	51.95	40.70	38.65
5000	481.60	270.05	270.00	77.00	57.05	55.65
6000	351.90	261.70	245.00	81.10	70.60	66.80
7000	390.90	293.30	283.40	121.70	95.00	88.60
8000	372.20	299.70	278.70	148.20	121.80	116.80
9000	659.90	386.50	336.40	343.50	191.50	167.30
10000	894.70	456.20	433.30	532.60	480.80	306.40
11000	667.00	413.20	355.50	614.40	413.20	365.80
12000	780.00	448.20	379.10	673.00	537.40	338.60
13000	582.80	386.50	314.70	753.90	548.60	443.70
14000	853.20	477.30	429.60	1022.70	769.00	539.30
15000	852.40	460.60	403.70	1543.00	841.70	742.30
16000	813.60	514.40	445.80	2070.00	922.70	837.80
17000	842.20	548.90	445.40	3921.60	1277.20	1064.20
18000	802.50	485.70	406.10	4204.70	1325.20	1077.00
19000	884.50	518.00	486.10	5171.50	1836.70	1685.50
20000	882.10	501.90	441.20	5703.70	2072.60	1568.30
21000	1064.30	638.00	564.90	7614.70	2526.40	2024.30
22000	985.10	607.60	495.10	7671.80	2537.60	2075.90
23000	826.30	597.70	463.20	7045.60	2516.40	1945.30
24000	1040.90	617.40	534.10	9969.20	3065.20	2840.80
25000	1114.70	617.20	580.30	11237.40	3067.50	2881.70

Com esses dados, particularmente os dados do cCG sem combinação afim, foi iniciada a validação do modelo. A equação (6.2) nos informa que o algoritmo cCG com 3 agentes realiza $2N^2 + 14N + 28$ operações por agente por iteração. Desprezando-se o tempo gasto com as somas entre escalares e com os pontos de sincronismo, podemos obter os dados de tempo médio por iteração, para cada dimensão de matriz. Utilizando-se a expressão $2N^2 + 14N + 28$, podemos avaliar quantas operações são feitas por agente por iteração, para cada dimensão de matriz. Dividindo-se o tempo médio por iteração pela quantidade de operações, podemos levantar o tempo médio por operação, conforme vemos na tabela 6.3.

Tabela 6.3: Dados para a validação do modelo do cCG com 3 agentes

Dimensão	**Operações por agente por iteração**	**Tempo médio por iteração (s)**	**Tempo médio por produto (ns)**
1000	2014028	0.008	3.80
1500	4521028	0.019	4.10
2000	8028028	0.030	3.72
2500	12535028	0.046	3.66
3000	18042028	0.069	3.83
3500	24549028	0.097	3.95
4000	32056028	0.124	3.86
4500	40563028	0.172	4.23
5000	50070028	0.211	4.22
6000	72084028	0.270	3.74
7000	98098028	0.324	3.30
8000	128112028	0.406	3.17
9000	162126028	0.495	3.06
10000	200140028	1.054	5.27
11000	242154028	1.027	4.24
12000	288168028	1.199	4.16
13000	338182028	1.419	4.20
14000	392196028	1.611	4.11
15000	450210028	1.827	4.06
16000	512224028	1.794	3.50
17000	578238028	2.327	4.02
18000	648252028	2.728	4.21
19000	722266028	3.546	4.91
20000	800280028	4.130	5.16
21000	882294028	3.960	4.49
22000	968308028	4.176	4.31
23000	1058322028	4.210	3.98
24000	1152336028	4.965	4.31
25000	1250350028	4.970	3.97

A quarta coluna da tabela deveria apresentar valores iguais, já que o tempo gasto por operação depende, em teoria, apenas do processador. Na prática, como desprezamos os tempos de soma e sobretudo os tempos para sincronismo, os dados de tempo médio por operação apresentam uma pequena diferença entre si, com uma média de 4.05 nanossegundos e com um desvio padrão de 0.50 nanossegundos, o que corresponde a 12.43% da média. Dessa maneira, podemos dizer que o tempo médio por iteração em função de N deveria ser $4.05 \cdot 10^{-9}(2N^2 + 14N + 28)$. Utilizando-se os dados reais de tempo médio por iteração, da terceira coluna da tabela 6.3, podemos ajustar uma parábola aos mesmos e comparar com essa parábola esperada. Isto pode ser observado na figura 6.1.

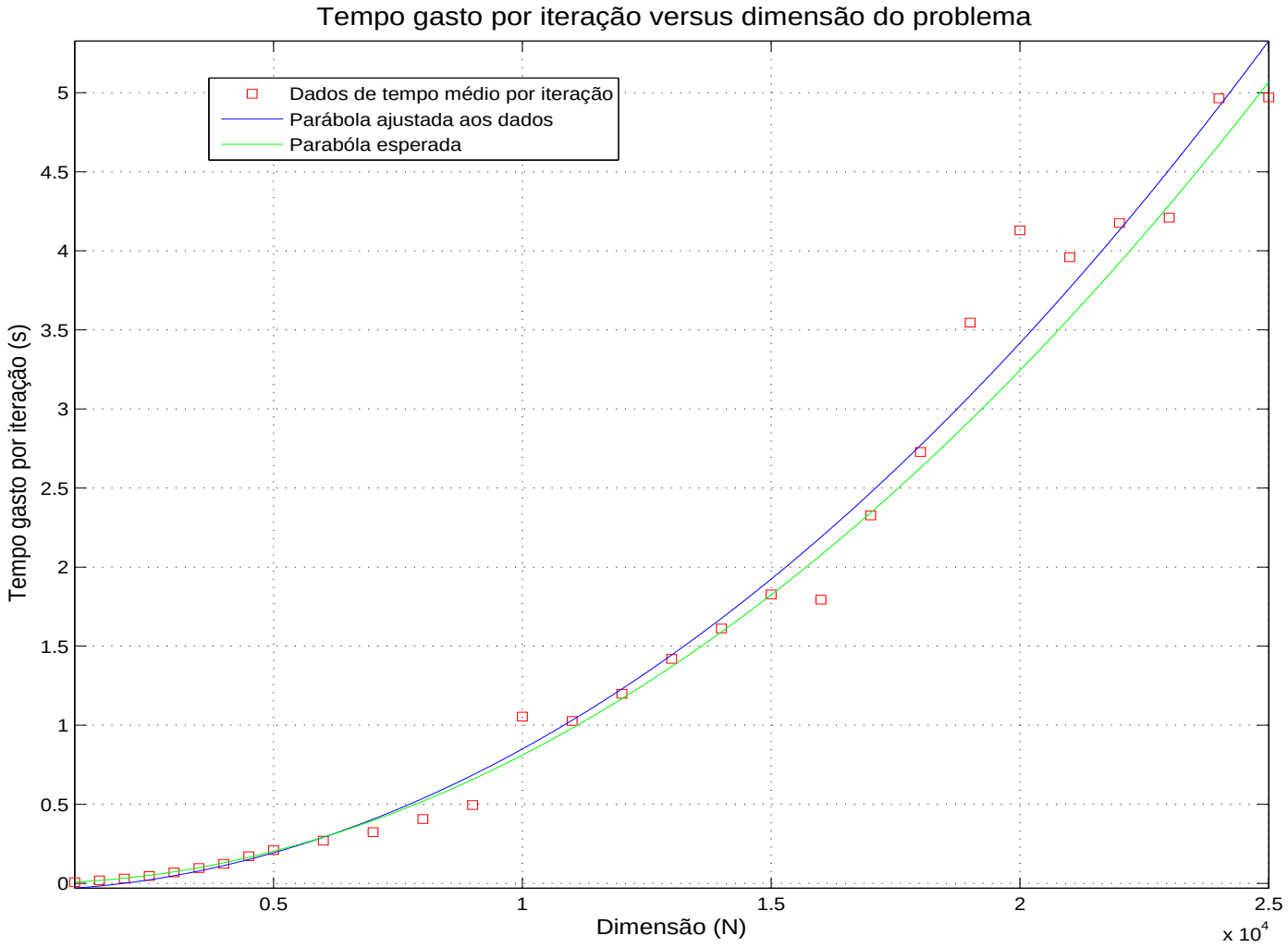

Figura 6.1: Validação do modelo de tempo médio por iteração do cCG

Nos quadrados vermelhos, vemos os dados originais de tempo médio por iteração. Em azul, vemos a parábola ajustada a esses dados por mínimos quadrados (erro quadrático médio = 0.04). Por fim, em verde, vemos a parábola esperada, isto é, $4.05 \cdot 10^{-9}(2N^2 + 14N + 28)$.

Primeiramente, analisando a qualidade da regressão pelo erro médio quadrático, é possível dizer que os dados de fato seguem uma parábola, conforme o esperado. Mais ainda, olhando para a comparação entre a parábola ajustada e a parábola esperada, vemos que ambas estão bastante próximas uma da outra, o que confirma o modelo de tempo por iteração obtido. A distância entre as parábolas aumenta com o aumento da dimensão N, justamente quando os tempos de cálculo desprezados passam a ter um efeito maior.

Em seguida, considerando que a complexidade do algoritmo cCG é de ordem $O(N^3)$, se observarmos os dados de tempo gasto para convergência (em escala log) por dimensão da matriz (também em escala log), é de se esperar uma reta com inclinação 3. Os dados supracitados e a parábola ajustada aos mesmos pode ser observada na figura 6.2.

Podemos ver nessa figura que os dados (em vermelho) seguem uma reta (em azul, ajustada por mínimos quadrados) com bastante precisão (o erro quadrático médio é 0.01). Além disso, a inclinação (2.542) é próxima de 3. O fato de tal inclinação ser menor que 3 se explica observando-se a precisão escolhida como critério de parada (10^{-3}). Essa precisão, embora produza soluções bastante acuradas, não é tão restritiva em relação a precisão de máquina disponível. Dessa maneira, o cCG acaba por convergir em menos de $\frac{N}{P}$ iterações. Além disso, o valor $\frac{N}{P}$ representa a quantidade de iterações para convergência do cCG no pior caso. Esses dois fatos, em conjunto, explicam a inclinação menor.

Da mesma maneira, podemos analisar a quantidade de iterações gastas para convergência em escala log-log. Uma vez que são esperadas $\frac{N}{3}$ iterações para a convergência, a inclinação esperada para a reta é 1. Esses dados são observados na figura 6.3. Em vermelho vemos os dados originais e em azul a reta ajustada. O erro médio quadrático também é pequeno (0.03), indicando que os dados de fato seguem uma reta; e a inclinação é menor que 1 (0.498). Esse fato é explicado novamente pela precisão não tão restritiva e pelo valor $\frac{N}{P}$ representar o pior caso de convergência.

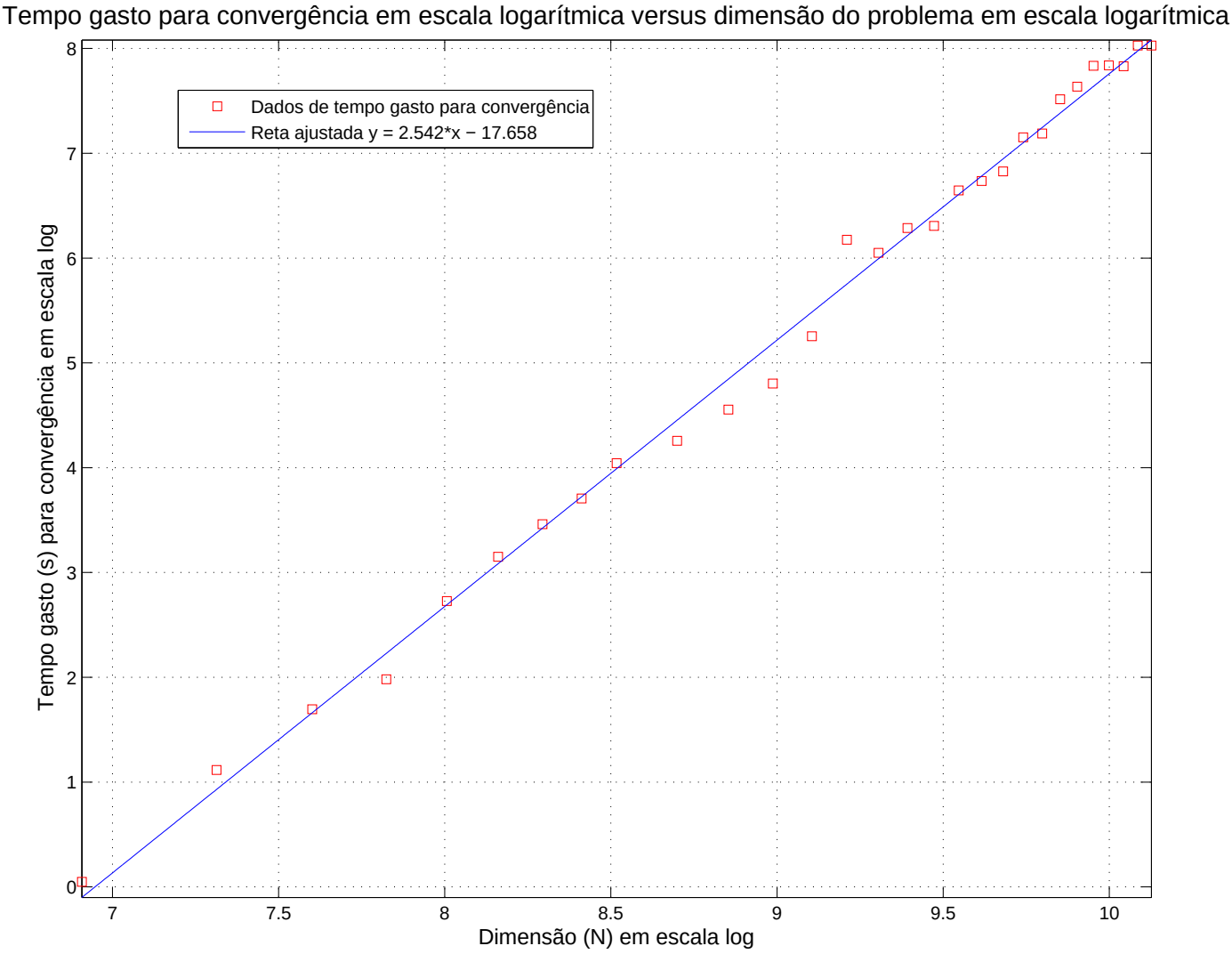

Figura 6.2: Validação do modelo de tempo total de convergência do cCG

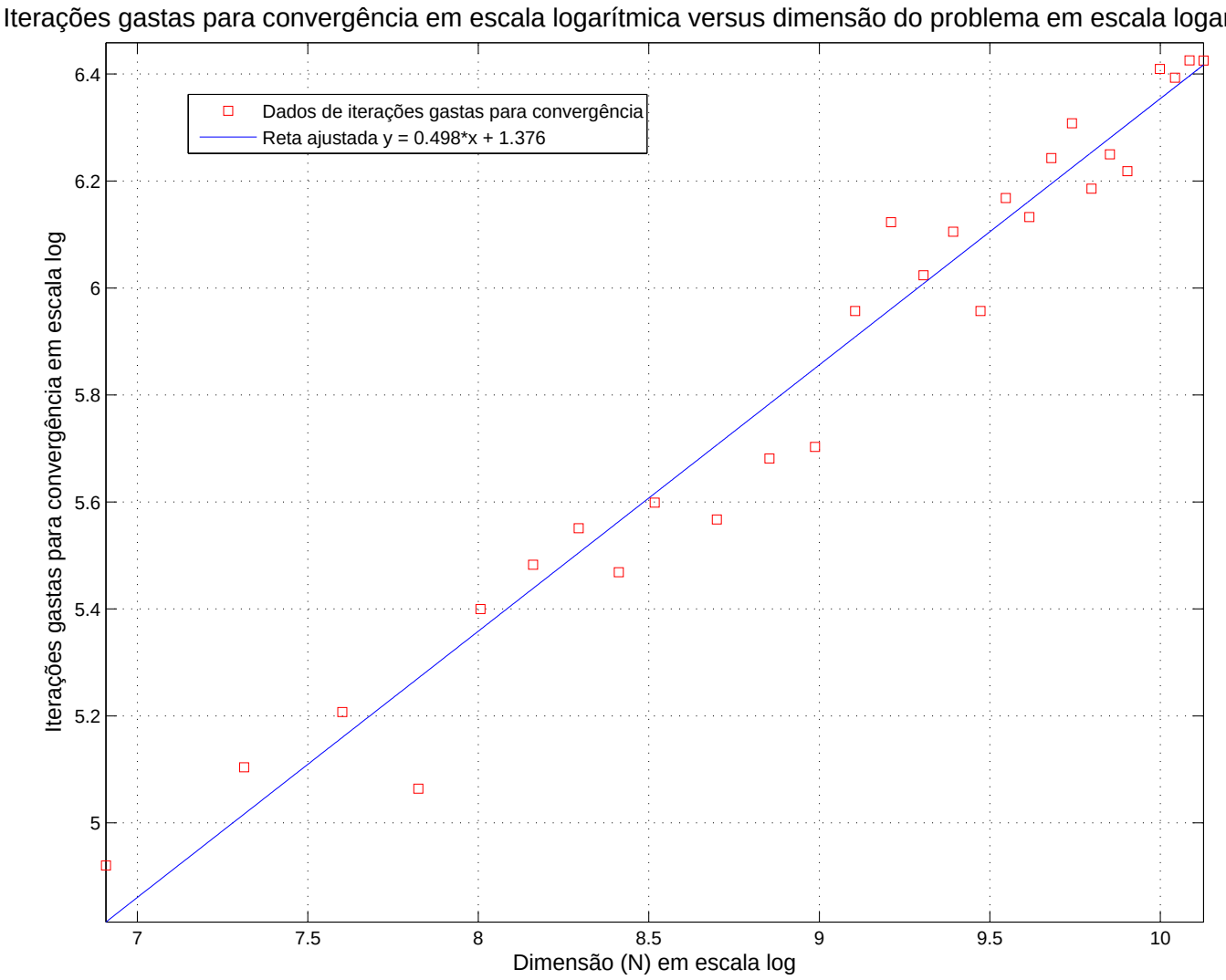

Figura 6.3: Validação do modelo de iterações gastas para convergência do cCG

Tendo em vista as figuras 6.1, 6.2 e 6.3, podemos afirmar que o comportamento do cCG foi, de fato, o esperado. Uma vez que o modelo está validado, faz-se necessária uma comparação entre o CG clássico, o cCG e o cCG com combinação afim. São utilizadas para tal comparação duas grandezas. Uma delas, o ***Speed-Up*** é definida como o tempo gasto pelo CG dividido pelo tempo gasto pelo cCG. A outra grandeza, nomeada **Ganho de Iterações**, é definida como a quantidade de iterações necessárias para a convergência do CG dividida pela quantidade de iterações gasta pelo cCG. Esses dados podem ser observados na tabela 6.4.

Tabela 6.4: Comparações entre o CG, o cCG e o cCG com combinação afim.

Dimensão	**Ganho de iterações CG-cCG**	***Speed-Up* CG-cCG**	**Ganho de iterações CG-acCG**	***Speed-Up* CG-acCG**
1000	1.79	1.71	1.86	1.80
1500	1.40	1.26	1.52	1.26
2000	1.66	1.48	1.77	1.63
2500	1.46	1.52	1.77	1.86
3000	1.57	1.35	1.58	1.39
3500	1.67	1.38	1.74	1.48
4000	1.55	1.27	1.67	1.31
4500	1.65	1.28	1.71	1.34
5000	1.78	1.35	1.78	1.38
6000	1.34	1.15	1.44	1.21
7000	1.33	1.28	1.38	1.37
8000	1.24	1.22	1.34	1.27
9000	1.80	1.79	2.07	2.05
10000	1.96	1.11	2.06	1.74
11000	1.61	1.45	1.88	1.68
12000	1.74	1.25	2.06	1.99
13000	1.51	1.37	1.85	1.70
14000	1.79	1.33	1.99	1.90
15000	1.85	1.83	2.11	2.08
16000	1.58	2.24	1.83	2.47
17000	1.53	3.07	2.07	3.64
18000	1.65	3.17	1.98	3.90
19000	1.71	2.82	1.82	3.07
20000	1.76	2.75	2.00	3.64
21000	1.67	3.01	1.88	3.76
22000	1.62	3.02	1.99	3.70
23000	1.38	2.80	1.78	3.62
24000	1.69	3.25	1.95	3.51
25000	1.81	3.66	1.92	3.90

É possível perceber um ganho significativo (no tempo e nas iterações) do cCG em relação ao CG, ainda maior quando incluída a combinação afim. Principalmente, esses ganhos são ainda maiores para dimensões mais elevadas (acima de $N = 16000$, aproximadamente). Esse fato pode ser explicado observando-se que o próprio CG apresenta essas propriedades, conforme já citado por Meurant e Strakoš em [9]. Isto é, para dimensões N pequenas a quantidade de iterações necessária para a convergência é menor que N, porém muito próxima de N. Para dimensões maiores, o CG acaba por levar muito menos do que N iterações para convergir, e os dados da tabela 6.4 levam a crer que o cCG incorporou essas propriedades.

As figuras 6.4 e 6.5 mostram essa tendência dos ganhos ao longo da dimensão N, juntamente com o ganho entre o cCG com combinação afim e sem combinação afim (que é apenas a razão entre os outros dois ganhos). É possível perceber que para dimensões elevadas o *Speed-Up* tende a 3, que é exatamente o seu valor esperado. O Ganho de iterações, no entanto, apresenta uma característica mais oscilatória, embora também seja maior para dimensões maiores.

Os valores médios do *Speed-Up* e do ganho de iterações CG-cCG são, respectivamente, 1.62 e 1.94. Os mesmos valores para a comparação CG-acCG são, respectivamente, 1.82 e 2.26. Essa média é reduzida pelos ganhos em dimensões menores mas, mesmo estes valores médios mostram a vantagem (em tempo e iterações) do cCG (com ou sem combinação afim) em relação ao CG.

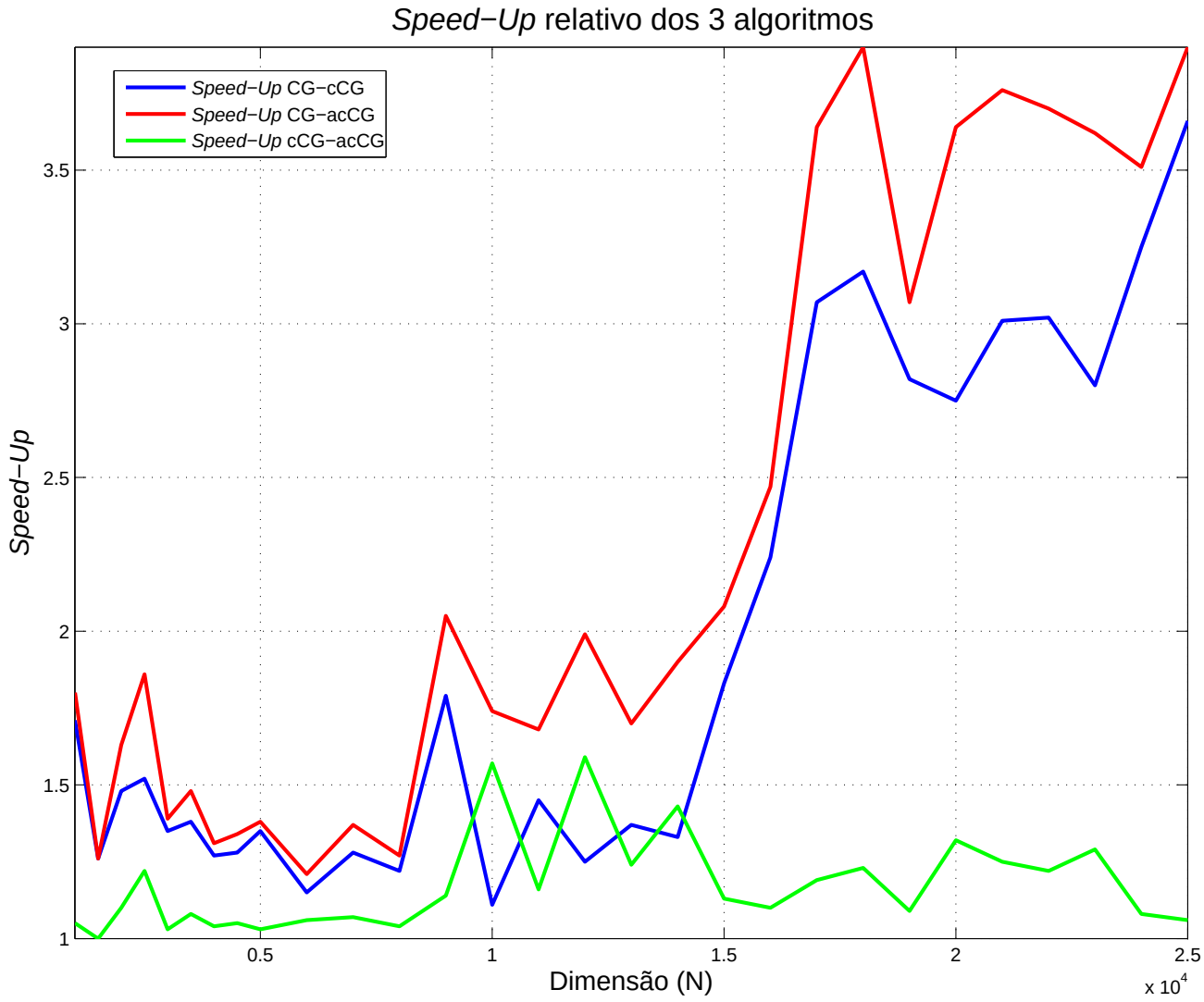

Figura 6.4: *Speed-Up* relativo do CG, cCG e acCG

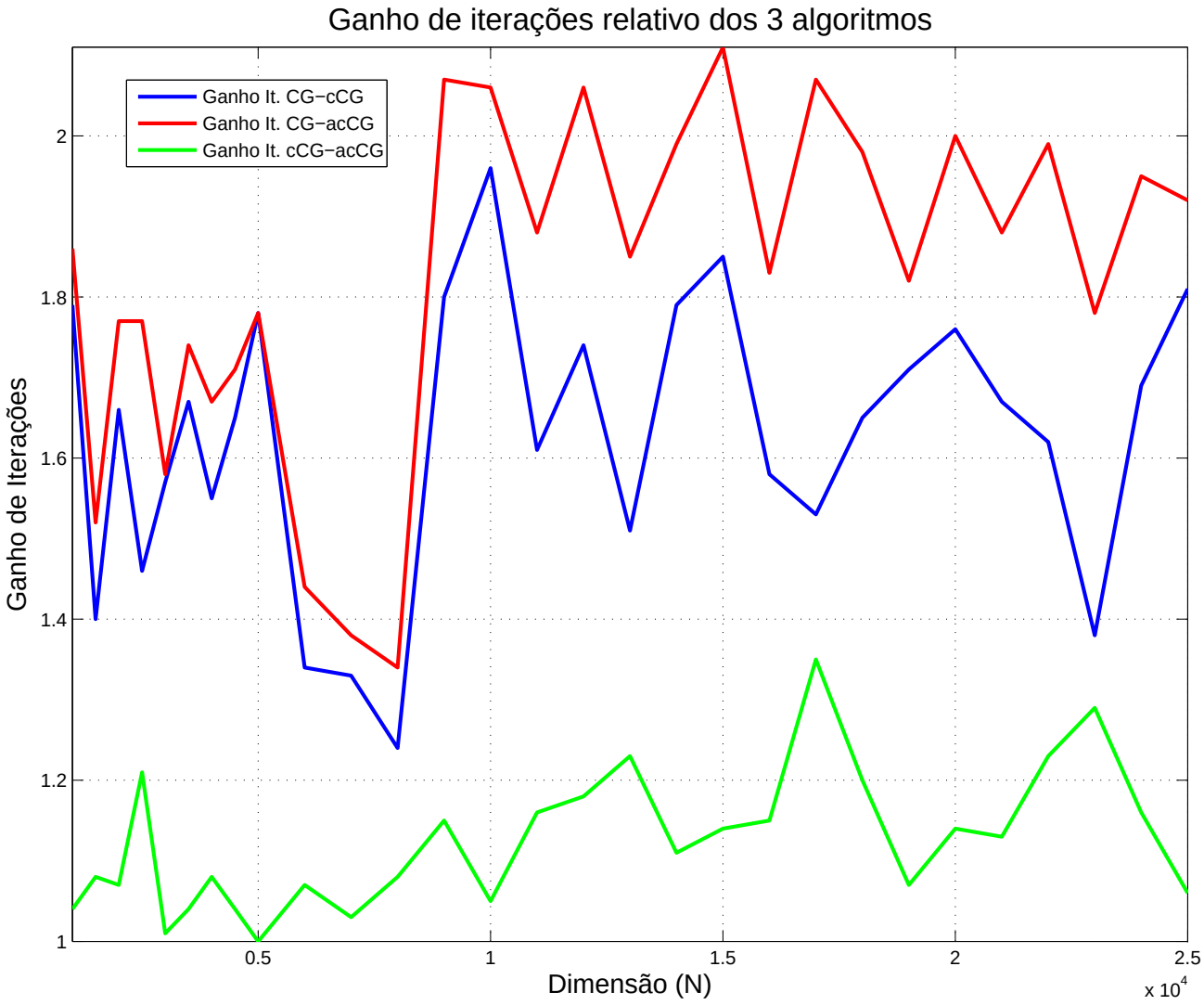

Figura 6.5: Ganho de iterações relativo do CG, cCG e acCG

Um outro experimento proposto foi relativo a tolerância do critério de parada. Utilizando uma matriz de dimensão $N = 8000$ e condição 10^6, a tolerância foi variada de 10^{-3} a 10^{-9}, rodando 10 condições iniciais para cada tolerância. Os resultados médios para o CG e para o cCG podem ser observados na tabela 6.5.

Tabela 6.5: Resultados médios para o CG e o cCG com diferentes tolerâncias.

Tolerância	**CG**		**cCG**	
	Tempo(s)	Iterações	Tempo(s)	Iterações
10^{-3}	148.20	372.20	121.80	299.70
10^{-4}	165.20	414.00	125.70	319.40
10^{-5}	177.40	444.90	132.00	335.00
10^{-6}	192.70	476.80	134.00	352.70
10^{-7}	206.50	510.70	135.20	336.20
10^{-8}	235.10	538.40	137.10	373.50
10^{-9}	275.00	559.70	141.90	381.20

É possível notar, conforme o esperado, que o tempo e o número de iterações necessários para convergência crescem com a diminuição da tolerância. Porém, um outro fato interessante, é que o Ganho de Iterações e o *Speed-Up* do cCG em relação ao CG também aumentam para tolerâncias mais restritivas. Esse fato é explicado se considerarmos que, ao rodar o algoritmo em uma tolerância próxima a precisão da máquina, estamos em uma situação similar a de precisão aritmética exata e, nesse caso, o Ganho de Iterações e o *Speed-Up* tenderiam a 3, que é a quantidade de agentes utilizada. Esse comportamento do Ganho de Iterações e do *Speed-Up* pode ser observado na figura 6.6.

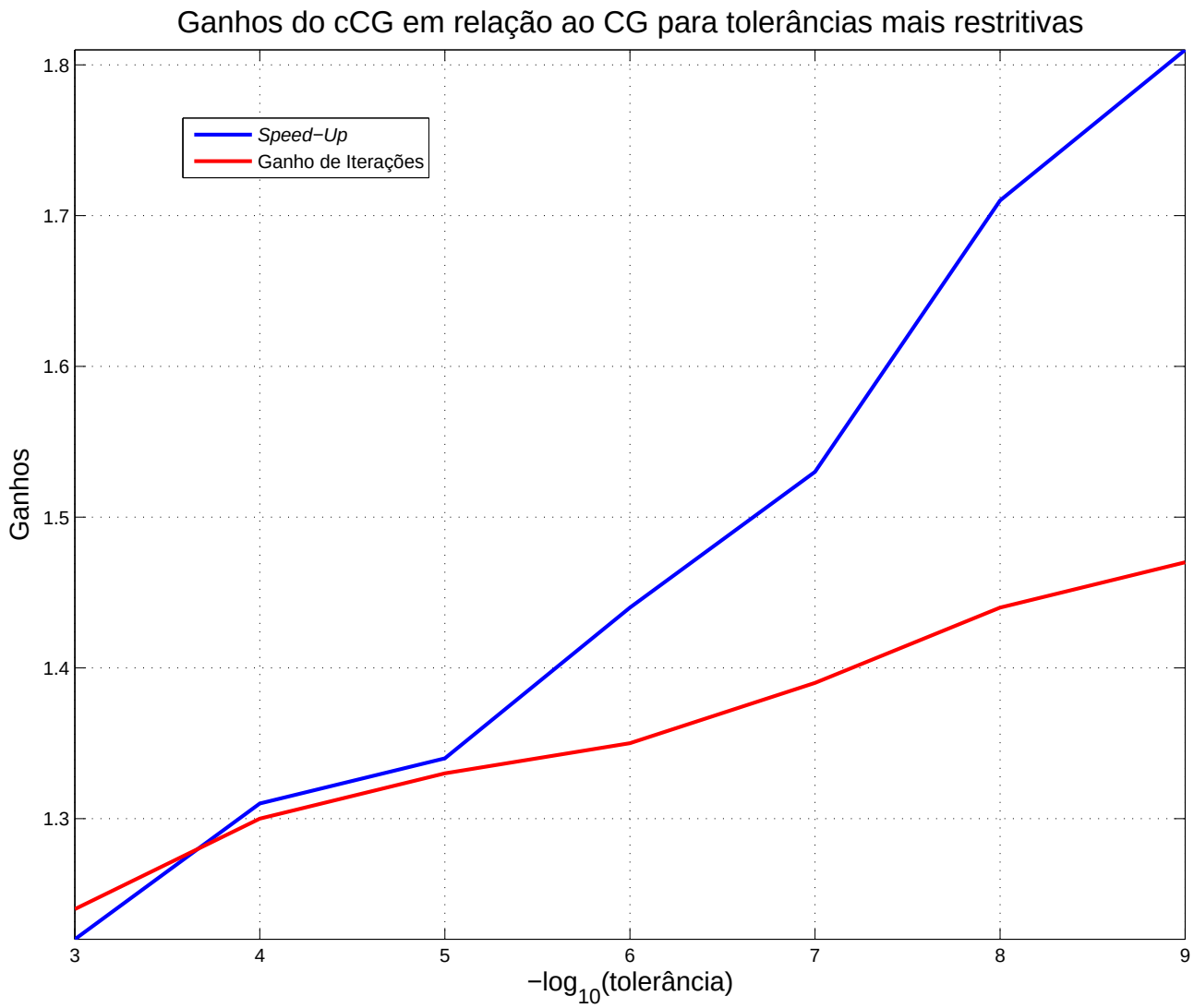

Figura 6.6: Aumento dos ganhos do cCG em relação ao CG para tolerâncias menores

O experimento com matrizes de Poisson, descrito por Demmel em [11] também foi testado. Matrizes de Poisson são esparsas e tem a forma observada em (6.3), isto é, 2 na diagonal, -1 na sub-diagonal e super-diagonal e 0 nas demais posições.

$$A = \begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & \dots & 0 & 0 \\ 0 & -1 & 2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 2 & -1 \\ 0 & 0 & 0 & \dots & -1 & 2 \end{bmatrix} \tag{6.3}$$

Foi utilizada uma matriz de Poisson de dimensão $N = 10000$. Conforme o esperado, para o CG tradicional houve convergência em exatas 10000 iterações, com tolerância de 10^{-9} pois, conforme citado por Demmel em [11], o resíduo se anula após exatas N iterações para matrizes de Poisson.

No entanto, para o cCG não houve convergência. Durante o processo iterativo, a matriz 3×3 do sistema que resolve α e β se torna singular. Isso porque todas as direções acabam convergindo para o mesmo vetor e, conseqüentemente, essa matriz (que é igual a $D_k^T A D_k$, sendo composta por todos os produtos direção-matriz-direção) apresenta o mesmo valor nas suas 9 posições.

Tal problema não ocorre com o CG clássico, pois o sistema que fornece α e β é de ordem 1×1, isto é, apenas uma divisão entre dois escalares, bastando que o lado esquerdo do sistema (denominador da divisão) seja não-nulo. Já para o cCG há um outro tipo de problema. Uma vez que $posto(D_k^T A D_k) = min(posto(D_k^T), posto(A), posto(D_k)) = posto(D_k)$, já que A tem posto completo, o posto da matriz $D_K^T A D_k$, que fornece α e β, é dado pelo posto de D_k. Assim, se duas ou mais direções de busca geradas na mesma iteração forem linearmente dependentes, D_k não terá posto completo e, conseqüentemente, $D_k^T A D_k$ também não o terá, sendo singular.

E tal fato ocorre justamente com matrizes A com autovalores próximos entre si (que é o caso da matriz de Poisson), onde as direções de busca acabam por convergir para o mesmo vetor, tornando impossível o cálculo dos passos α e β, mostrando que o algoritmo cCG da forma proposta neste trabalho deveria sofrer modificações quando aplicado a matrizes com essas distribuições de autovalores.

Capítulo 7

Conclusões e Trabalhos Futuros

Deste trabalho, é possível concluir que o algoritmo proposto (Gradiente Conjugado Cooperativo) incorporou para si diversas características do método Gradiente Conjugado tradicional. A sua convergência se dá em norma-A^{-1} decrescente do resíduo, enquanto a norma euclidiana oscila, mas acaba por convergir também, assim como no CG.

Foi possível, também, validar o modelo de tempo e iterações gastas para convergência do cCG, verificando-se a quantidade de operações realizadas. O modelo de quantidade de operações por iteração também segue o padrão do CG. No entanto, foi possível notar a convergência mais rápida do cCG. Este ponto é o mais notável deste trabalho, tendo sido possível encontrar uma gama de matrizes (sem autovalores agrupados e razoavelmente bem condicionadas) em que o cCG converge mais rápido que o CG. Adicionalmente, essa rapidez cresce para dimensões maiores. Ao se escolher uma tolerância mais restritiva, os ganhos de tempo e quantidade de iterações também aumentam, o que é um resultado interessante. O principal fato que se pode concluir deste trabalho é que foi possível obter ganhos de desempenho em relação ao algoritmo mais utilizado para solução de sistemas lineares de grandes dimensões.

Para matrizes mal-condicionadas, o desempenho do cCG é mais afetado que o CG clássico. Para matrizes com distribuição de autovalores desfavorável, o desempenho do cCG também não é bom. Esses resultados, embora negativos, são importantes para definir as desvantagens do uso do cCG em detrimento do CG, a fim de conhecer quando é vantajoso utilizar o algoritmo cCG. Todavia, para trabalhos futuros é necessário encontrar uma forma de obter bons desempenhos para matrizes com tal distribuição de autovalores.

Justamente pelo fato de os ganhos aumentarem para dimensões maiores, um dos desafios futuros relativos a este trabalho é rodar o algoritmo para dimensões ainda maiores que 25000. Não foi possível realizar tais testes devido a limitações de memória, mesmo utilizando-se *clusters*, pois as matrizes utilizadas são cheias, apresentando uma elevada quantidade de elementos a armazenar.

Um outro ponto a se compreender melhor é o desempenho em aritmética de precisão finita do cCG. É bem sabido que, embora existam estudos de precisão finita para o CG (por exemplo o exposto em [9]), o seu comportamento em precisão finita ainda não é completamente conhecido. Essa busca pelo entendimento do comportamento em precisão finita também se aplica ao cCG. Ainda nesse âmbito, deve-se entender melhor o efeito da combinação afim. Embora tenha sido notado neste trabalho que a mesma tem um efeito positivo para o cCG, provavelmente por questões numéricas, é necessário compreender melhor os seus efeitos no cCG.

Por fim, como um outro trabalho a se realizar futuramente, destaca-se o uso de pré-condicionadores. Os efeitos dos mesmos sobre o CG são conhecidos e espera-se que eles apresentem um bom resultado quando combinados com o cCG. Isto é uma outra questão que necessita de mais investigação.

Referências Bibliográficas

[1] KUMAR, V., LEONARD, N., MORSE, A. S. (Eds.). *2003 Block Island Workshop on Cooperative Control*, Lecture Notes in Control and Information Sciences, vol. 309, 2005. Springer. ISBN: 978-3-540-22861-5.

[2] NEDIC, A., OZDAGLAR, A. "Convex Optimization in Signal Processing and Communications". cap. Cooperative Distributed Multi-agent Optimization, pp. 340–386, Cambridge University Press, 2010.

[3] MURRAY, R. M. "Recent research in cooperative control of multi-vehicle systems", *J. Guidance, Control and Dynamics*, v. 129, n. 5, pp. 571–583, September 2007.

[4] PENNY, N. "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures", *The AI Magazine*, 1996.

[5] DUSONCHET, Y., TALUKDAR, S., SINNOT, H. "Load Flows Using a Combination of Point Jacobi and Newton's Methods", *IEEE Trans. Power Aparatus and Systems*, v. 90, pp. 941–949, 1996.

[6] BREZINSKI, C., REDIVO-ZAGLIA, M. "Hybrid procedures for solving linear systems", *Numerische Mathematik*, 1994.

[7] BHAYA, A., BLIMAN, P.-A., PAZOS, F. "Cooperative parallel asynchronous computation of the solution of symmetric linear systems". In: *Proc. of the 49th IEEE Conference on Decision and Control*, Atlanta, USA, December 2010.

[8] HESTENES, M. R., STIEFEL, E. "Methods of Conjugate Gradients for Solving Linear Systems", *Journal of Research of the Natural Bureau of Standards*, v. 49, n. 6, dez. 1952.

[9] MEURANT, G., STRAKOŠ, Z. "The Lanczos and conjugate gradient algorithms in finite precision arithmetic", *Acta Numerica*, 2006.

[10] GREENBAUM, A. *Iterative methods for solving linear systems*. Philadelphia, SIAM, 1997.

[11] DEMMEL, J. *Applied Numerical Linear Algebra*. 3rd ed. Philadelphia, Society for Industrial and Applied Mathematics, 3600 University City Science Center, 1996.

[12] TREFETHEN, L. N., BAU, D. *Numerical Linear Algebra*. Philadelphia, Society for Industrial and Applied Mathematics, 3600 University City Science Center, 1997.

[13] BHAYA, A., KASZKUREWICZ, E. *Control perspectives on numerical algorithms and matrix problems*. Advances in Control. Philadelphia, SIAM, 2006.

[14] TRAUB, J. F., WOŹNIAKOWSKI, H. "On the Optimal Solution of Large Linear Systems", *Journal of the Association for Computing Machinery*, v. 31, n. 3, pp. 545–559, 1984.

[15] SHEWCHUK, J. "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain", *School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213*, ago. 1994.

[16] LUENBERGER, D. G., YE, Y. *Linear and nonlinear programming*. Reading, MA, Addison-Wesley, 1984. ISBN: 0-201-15794-2.

[17] STRANG, G. *Linear Algebra and its Applications*. 3rd ed. New York, Harcourt Brace Jovanovich, Inc., 1998.

[18] GÜLER, O. *Foundations of Optimization*. New York, Springer, 2010. ISBN: 978-0-387-34431-7.

[19] BHAYA, A., KASZKUREWICZ, E. "Iterative methods as dynamical systems with feedback control." *42nd Conference on Decision and Control - CDC*, pp. 545–559, 2003.

[20] SHILON, O. "RandOrthMat.m: MATLAB code to generate a random $n \times n$ orthogonal real matrix". 2006. Disponível em: <`http://www.mathworks.com/matlabcentral/fileexchange/authors/23951`>.

[21] "Códigos para o CG, cCG, cCG afim e para a geração de matrizes positivas definidas". March 2012. Disponível em: <`http://www.nacad.ufrj.br/~amit`>.

[22] "Matrix Market". June 2011. Disponível em: <`http://math.nist.gov/MatrixMarket/`>.

Apêndice A

Códigos utilizados

A.1 Código do algoritmo CG clássico

```
/*
 * Program: CG.c (main routine)
 *
 * C code by Guilherme da Silva Niedu
 * Rio de Janeiro, Jun 14 2011
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "ccomputing.h" // includes function headers for cooperative
   computing
#include "mmio.h"

int main(int argc, char *argv[])
{
    // Declare algorithm variables
    int P; // communicate at each P iterations
    int m, n, i, j, k; // matrix dimension and the row and column
       indices
    double T, exchangeDir; //T, exchangeDir = random numbers, P=
       probability
    double tol; // tolerance for the stop criterion

    int ret_code;
    MM_typecode matcode;
```

```
int M, N, nz;
int *I, *J;
double *A;

// Files
FILE *f;
FILE *rhsfp;
FILE *icXfp;
FILE *initCond; // matrix with all initial conditions
FILE *reportFP;
FILE *iterFP;
FILE *solAgXfp;

// File names
char report[30];
char iterations[30];
char mmfFile[30];
char rhsFile[30];
char icXFile[30];

// Maximum number of iterations
int maxIterNum;


// Check validity of the command line input arguments
if (argc < 9)
{
   fprintf(stderr, "Usage: %s -mmf [matrix-market-filename] -rhs [
      righthand side file] -icX [x0] -icY [y0] -icZ [z0] -tol [tol]
      -comm [number of iterations] -agX [agX] -agY [agY] -agZ [agZ
      ] -repfile [output-report-filename] -iterfile [file to save
      number of iterations] -maxiternum [max number of iterations]
      -numAgs [number of agents]\n", argv[0]);
   exit(1);
}

// Loads data from commandline input
for ( i=1; i<argc; i+=2 )
{

  // Matrix market file name
  if ( strcmp(argv[i],"-mmf")==0 )
  {
    f = fopen(argv[i+1],"r");
    strcpy(mmfFile,argv[i+1]);
```

```
}

// Right hand side file name
if ( strcmp(argv[i],"-rhs")==0 )
{
  rhsfp = fopen(argv[i+1],"r");
  strcpy(rhsFile,argv[i+1]);
}

// Initial condition for agent X
if ( strcmp(argv[i],"-icX")==0 )
{
  icXfp = fopen(argv[i+1],"r");
  strcpy(icXFile,argv[i+1]);
}


// Tolerance
if ( strcmp(argv[i],"-tol")==0 )
{
  tol = atof(argv[i+1]);
}

// Report file
if ( strcmp(argv[i],"-repfile")==0 )
{
  reportFP = fopen(argv[i+1],"w");
  strcpy(report,argv[i+1]);
}

// File with the number of iterations
// it will contain the number of iterations over
// several runs of the algorithm for several initial conditions
if ( strcmp(argv[i],"-iterfile")==0 )
{
  iterFP = fopen(argv[i+1],"a");
  strcpy(iterations,argv[i+1]);
}

// Specify maximum number of iterations
if ( strcmp(argv[i],"-maxiternum")==0 )
{
  maxIterNum = atoi(argv[i+1]);
}
```

```
} // end for(i=1;i<argc;i+=2)

// Checks if the max number of iterations was informed
// otherwise, set it to 10000
if (maxIterNum==0)
  maxIterNum=10000;
// Loads matrix into memory in matrix market form
if (mm_read_banner(f, &matcode) != 0)
{
    printf("Could not process Matrix Market banner.\n");
    exit(1);
}

//  This is how one can screen matrix types if their application
//  only supports a subset of the Matrix Market data types.
if (mm_is_complex(matcode) && mm_is_matrix(matcode) &&
        mm_is_sparse(matcode) )
{
    printf("Sorry, this application does not support ");
    printf("Market Market type: [%s]\n", mm_typecode_to_str(matcode
       ));
    exit(1);
}

// find out size of sparse matrix ....
if ((ret_code = mm_read_mtx_crd_size(f, &M, &N, &nz)) !=0)
    exit(1);


// reseve memory for matrices
I = (int *) malloc(nz * sizeof(int));
J = (int *) malloc(nz * sizeof(int));
A = (double *) malloc(nz * sizeof(double)); // coefficients matrix


// NOTE: when reading in doubles, ANSI C requires the use of the "l
   "
//   specifier as in "%lg", "%lf", "%le", otherwise errors will
   occur
//  (ANSI C X3.159-1989, Sec. 4.9.6.2, p. 136 lines 13-15)

for (i=0; i<nz; i++)
{
    fscanf(f, "%d %d %lg\n", &I[i], &J[i], &A[i]);
    I[i]--;  // adjust from 1-based to 0-based
```

```
    J[i]--;
}

if (f !=stdin) fclose(f);

// Set values for m and n (rows and columns)
m = M;
n = N;


// Current state vectors
double *X = (double*)malloc(n*sizeof(double));

//Alpha and Beta steps
double alpha;
double beta;

//Form alpha and beta
double den;

// Product of matrix A by vetor x
double *AX = (double*)malloc(n*sizeof(double));


// Residuals for each agent
double *RX = (double*)malloc(n*sizeof(double));

// Directions for each agent
double *DX = (double*)malloc(n*sizeof(double));

//Products matrix-vector to form alpha and beta
double *ADX = (double*)malloc(n*sizeof(double));


// Loads initial conditions
char lineX[20];
i=0;
while(fgets(lineX,20,icXfp) != NULL)
{
     X[i] = atof(lineX);
     i++;
}
fclose(icXfp);

// Loads righthand side of Ax=b
```

```
char line[20];
i=0;
double *b = (double*)malloc(n*sizeof(double));
while(fgets(line,20,rhsfp) != NULL)
{
     b[i] = atof(line);
     i++;
}
fclose(rhsfp);

//Computes product A*X
matvecsr_smm(A, I, J, X, n, nz, AX);

//Computes the residues relative to the initial conditions
for(j=0;j<n;j++)
{
  RX[j] = AX[j]-b[j]; //calculate residue value for X
}

//Computes the directions relative to the initial conditions
copyVect(RX,DX,n); //calculate direction for X

// Stop criterion
double minRes;

// Generates a seed for random numbers
srand(time(NULL));  //Generate a seed

// Count time
time_t t0, t1;

// Main loop of cooperative computation with probability P
int iterNum=0;
t0 = clock(); // starts counting time

do // starts do-while loop
{
  // Increments iteration number counter
  iterNum++;

  matvecsr_smm(A, I, J, DX, n, nz, ADX);

  //Calculates alpha
  den = dotprod(DX,ADX,n);
  alpha = (-1)*dotprod(RX,DX,n)/den;
```

```
  //Updates all agents
  for(i=0;i<n;i++)
  {
    X[i] = X[i] + alpha*DX[i];
  }

  //Computes product A*X
  matvecsr_smm(A, I, J, X, n, nz, AX);

  //Updates all residues
  for(i=0;i<n;i++)
  {
    RX[i] = AX[i] - b[i];
  }

  //Calculates beta
  beta = (-1)*dotprod(RX,ADX,n)/den;

  //Updates all directions
  for(i=0;i<n;i++)
  {
    DX[i] = RX[i] + beta*DX[i];
  }



  //Computes residues norms
  minRes = norm(RX,n);



  // Saves minimum residual norm
  //fprintf(fminres,"%d %f\n",iterNum, minRes);

}
while ( minRes > tol && iterNum < maxIterNum ); //end while

// Ends time count
t1 = clock();

// Open files to store residues
FILE *fminres;
char minresFN[20];
strcpy(minresFN,mmfFile);
```

```
strcat(minresFN,"mrn");
fminres= fopen(minresFN,"w");

// Open files to save the solution
solAgXfp = fopen("solAgX.txt","w");


//Saves report file


// Date and time
time_t now;
time(&now);
fprintf(reportFP,"%s\n", ctime(&now));

fprintf(reportFP,"Execution␣report␣of␣program␣%s\n\n", argv[0]);

// Input files
fprintf(reportFP,"INPUT␣FILES\n");
fprintf(reportFP,"=====␣=====\n\n");
fprintf(reportFP,"Matrix␣A␣(matrix␣market␣format):␣%s\n",mmfFile);
fprintf(reportFP,"Right-hand␣side␣vector␣b:␣%s\n",rhsFile);
fprintf(reportFP,"Initial␣condition␣vector:␣%s\n",icXFile);

// Input variables
fprintf(reportFP,"\nINPUT␣VARIABLES\n");
fprintf(reportFP,"=====␣=========\n\n");
fprintf(reportFP,"Agent␣running␣Conjugate␣Gradient\n");

fprintf(reportFP,"Tolerance␣(stop␣rule):␣%e\n", tol);

// Solution vector
fprintf(reportFP,"\nSOLUTION␣VECTOR\n");
fprintf(reportFP,"========␣======\n\n");


if ( minRes <= tol )
{
  fprintf(reportFP,"Error:␣%e\n", minRes);
  fprintf(reportFP,"Solution␣found␣in␣%d␣iterations␣within␣a␣
     tolerance␣of␣%e␣\n\n",iterNum,tol);
}
else
{
  fprintf(reportFP,"Algorithm␣did␣not␣satisfy␣the␣stop␣rule␣(error␣
```

```
      <= %e ) after %d iterations.\n\n",tol, iterNum);
  fprintf(reportFP,"Error after %d iterations: %e\n\n",iterNum,
      minRes);
}

fprintf(reportFP,"Total running time: %.3f seconds\n",(float)((t1-
    t0)/CLOCKS_PER_SEC));


for (i=0;i<n;i++)
{
  fprintf(reportFP,"x(%d) = %f \n",i,X[i]);

  // Solutions to separate files
  fprintf(solAgXfp,"%f\n",X[i]);
}
fprintf(reportFP,"\n---------------------------- End of Report
    ----------------------------\n");

// Closes report file
fclose(reportFP);

// Saves number of iterations to file a separate file
fprintf(iterFP,"%i\n",iterNum);

// Closes iterations file
fclose(iterFP);

// Send output to screen
printf("\nProgram execution completed after %d iterations...\n",
    iterNum);
printf("Total running time: %.3f seconds\n",(float)((t1-t0)/
    CLOCKS_PER_SEC));
printf("\nFiles saved:\n");
printf("%s - program execution report\n",report);
printf("%s - record of number of iterations over several runs\n",
    iterations);
printf("solAgX.txt - solution computed by the program\n");
printf("\n");

//////////////////
// Close files
fclose(fminres);

// End program
```

```
    return 0;
}
```

A.2 Código do algoritmo cCG sem combinação afim

```
/*
 * Program: cCG.c (main routine)
 *
 * C code by Guilherme da Silva Niedu
 * Rio de Janeiro, Jun 14 2011
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "ccomputing.h" // includes function headers for cooperative
   computing
#include "mmio.h"
#include "omp.h"

int main(int argc, char *argv[])
{
    // Declare algorithm variables
    int P, F; // communicate at each P iterations
    int m, n, i, j, k; // matrix dimension and the row and column
       indices
    double T, exchangeDir; //T, exchangeDir = random numbers, P=
       probability
    double tol; // tolerance for the stop criterion

    int ret_code;
    MM_typecode matcode;


    int M, N, nz;
    int *I, *J;
    double *A;

    // Files
    FILE *f;
    FILE *rhsfp;
    FILE *icXfp;
    FILE *icYfp;
    FILE *icZfp;
    FILE *initCond; // matrix with all initial conditions
    FILE *reportFP;
    FILE *iterFP;
    FILE *solAgXfp;
```

```
FILE *solAgYfp;
FILE *solAgZfp;

// File names
char report[30];
char iterations[30];
char mmfFile[30];
char rhsFile[30];
char icXFile[30];
char icYFile[30];
char icZFile[30];

// Number of agents
int numAgents = 3; // default value, it will be overwritten if -
    numAgs is present in command line arguments
// Vector of agent values
int agentVal[numAgents];
// Maximum number of iterations
int maxIterNum;


// Check validity of the command line input arguments
if (argc < 23)
{
    fprintf(stderr, "Usage: %s -mmf [matrix-market-filename] -rhs [
        righthand side file] -icX [x0] -icY [y0] -icZ [z0] -tol [tol]
        -comm [number of iterations] -agX [agX] -agY [agY] -agZ [agZ
        ] -repfile [output-report-filename] -iterfile [file to save
        number of iterations] -maxiternum [max number of iterations]
        -numAgs [number of agents]\n", argv[0]);
    exit(1);
}

// Loads data from commandline input
for ( i=1; i<argc; i+=2 )
{

  // Matrix market file name
  if ( strcmp(argv[i],"-mmf")==0 )
  {
    f = fopen(argv[i+1],"r");
    strcpy(mmfFile,argv[i+1]);
  }

  // Right hand side file name
```

```
if ( strcmp(argv[i],"-rhs")==0 )
{
  rhsfp = fopen(argv[i+1],"r");
  strcpy(rhsFile,argv[i+1]);
}

// Initial condition for agent X
if ( strcmp(argv[i],"-icX")==0 )
{
  icXfp = fopen(argv[i+1],"r");
  strcpy(icXFile,argv[i+1]);
}

// Initial condition for agent Y
if ( strcmp(argv[i],"-icY")==0 )
{
  icYfp = fopen(argv[i+1],"r");
  strcpy(icYFile,argv[i+1]);
}

// Initial condition for agent Z
if ( strcmp(argv[i],"-icZ")==0 )
{
  icZfp = fopen(argv[i+1],"r");
  strcpy(icZFile,argv[i+1]);
}

// Probability
if ( strcmp(argv[i],"-comm")==0 )
{
  P = atoi(argv[i+1]);
}

// First Communication
if ( strcmp(argv[i],"-fcomm")==0 )
{
  F = atoi(argv[i+1]);
}

// Number of agents
if ( strcmp(argv[i],"-numAgs")==0 )
{
  numAgents = atoi(argv[i+1]);
}
```

```
  // Tolerance
  if ( strcmp(argv[i],"-tol")==0 )
  {
    tol = atof(argv[i+1]);
  }

  // Report file
  if ( strcmp(argv[i],"-repfile")==0 )
  {
    reportFP = fopen(argv[i+1],"w");
    strcpy(report,argv[i+1]);
  }

  // File with the number of iterations
  // it will contain the number of iterations over
  // several runs of the algorithm for several initial conditions
  if ( strcmp(argv[i],"-iterfile")==0 )
  {
    iterFP = fopen(argv[i+1],"a");
    strcpy(iterations,argv[i+1]);
  }

  // Specify maximum number of iterations
  if ( strcmp(argv[i],"-maxiternum")==0 )
  {
    maxIterNum = atoi(argv[i+1]);
  }

} // end for(i=1;i<argc;i+=2)

// Checks if the max number of iterations was informed
// otherwise, set it to 10000
if (maxIterNum==0)
  maxIterNum=10000;
// Loads matrix into memory in matrix market form
if (mm_read_banner(f, &matcode) != 0)
{
    printf("Could␣not␣process␣Matrix␣Market␣banner.\n");
    exit(1);
}

//  This is how one can screen matrix types if their application
//  only supports a subset of the Matrix Market data types.
if (mm_is_complex(matcode) && mm_is_matrix(matcode) &&
        mm_is_sparse(matcode) )
```

```
{
    printf("Sorry, this application does not support ");
    printf("Market Market type: [%s]\n", mm_typecode_to_str(matcode
        ));
    exit(1);
}

// find out size of sparse matrix ....
if ((ret_code = mm_read_mtx_crd_size(f, &M, &N, &nz)) !=0)
    exit(1);


// reseve memory for matrices
I = (int *) malloc(nz * sizeof(int));
J = (int *) malloc(nz * sizeof(int));
A = (double *) malloc(nz * sizeof(double)); // coefficients matrix


// NOTE: when reading in doubles, ANSI C requires the use of the "l
    "
//   specifier as in "%lg", "%lf", "%le", otherwise errors will
    occur
//  (ANSI C X3.159-1989, Sec. 4.9.6.2, p. 136 lines 13-15)


for (i=0; i<nz; i++)
{
    fscanf(f, "%d %d %lg\n", &I[i], &J[i], &A[i]);
    I[i]--;  // adjust from 1-based to 0-based
    J[i]--;
}

if (f !=stdin) fclose(f);

// Set values for m and n (rows and columns)
m = M;
n = N;


// Current state vectors
double *X = (double*)malloc(n*sizeof(double));
double *Y = (double*)malloc(n*sizeof(double));
double *Z = (double*)malloc(n*sizeof(double));

//Alpha and Beta steps
```

```
double alpha[numAgents][numAgents];
double beta[numAgents][numAgents];

// Old state vectors

// Product of matrix A by vetor x
double *AX = (double*)malloc(n*sizeof(double));
// Product of matrix A by vetor y
double *AY = (double*)malloc(n*sizeof(double));
// Product of matrix A by vetor z
double *AZ = (double*)malloc(n*sizeof(double));


// Residuals for each agent
double *RX = (double*)malloc(n*sizeof(double));
double *RY = (double*)malloc(n*sizeof(double));
double *RZ = (double*)malloc(n*sizeof(double));

// Directions for each agent
double *DX = (double*)malloc(n*sizeof(double));
double *DY = (double*)malloc(n*sizeof(double));
double *DZ = (double*)malloc(n*sizeof(double));

//New Directions for each agent
double *NDX = (double*)malloc(n*sizeof(double));
double *NDY = (double*)malloc(n*sizeof(double));
double *NDZ = (double*)malloc(n*sizeof(double));

//Matrix MAT, to compute alpha and beta
double MAT[numAgents*numAgents];
double MAT_original[numAgents*numAgents];


//Products matrix-vector to form MAT
double *ADX = (double*)malloc(n*sizeof(double));
double *ADY = (double*)malloc(n*sizeof(double));
double *ADZ = (double*)malloc(n*sizeof(double));

//Elements of MAT
double mat11, mat12, mat13, mat22, mat23, mat33;

//Right-hand side for systems involving MAT
double *MAT_b = (double*)malloc(numAgents*numAgents*sizeof(double))
    ;
```

```
//Matrix needed for communitacion step
double COMM[(numAgents-1)*(numAgents-1)];

//Elements of comm
double comm11, comm12, comm22;

//Vectors needed for communitacion step
double COMM_b[(numAgents-1)];
double *rho_1 = (double*)malloc(n*sizeof(double));
double *rho_2 = (double*)malloc(n*sizeof(double));
double *sigma_1 = (double*)malloc(n*sizeof(double));
double *sigma_2 = (double*)malloc(n*sizeof(double));
double lambda[numAgents];

// Loads initial conditions
char lineX[20];
i=0;
while(fgets(lineX,20,icXfp) != NULL)
{
     X[i] = atof(lineX);
     i++;
}
fclose(icXfp);

char lineY[20];
i=0;
while(fgets(lineY,20,icYfp) != NULL)
{
     Y[i] = atof(lineY);
     i++;
}
fclose(icYfp);

char lineZ[20];
i=0;
while(fgets(lineZ,20,icZfp) != NULL)
{
     Z[i] = atof(lineZ);
     i++;
}
fclose(icZfp);

// Loads righthand side of Ax=b
char line[20];
i=0;
```

```
double *b = (double*)malloc(n*sizeof(double));
while(fgets(line,20,rhsfp) != NULL)
{
     b[i] = atof(line);
     i++;
}
fclose(rhsfp);

//Computes product A*X, A*Y and A*Z
matvecsr_smm(A, I, J, X, n, nz, AX);
matvecsr_smm(A, I, J, Y, n, nz, AY);
matvecsr_smm(A, I, J, Z, n, nz, AZ);

//Computes the residues relative to the initial conditions
for(j=0;j<n;j++)
{
  RX[j] = AX[j]-b[j]; //calculate residue value for X
  RY[j] = AY[j]-b[j]; //calculate residue value for Y
  RZ[j] = AZ[j]-b[j]; //calculate residue value for Z
}

//Computes the directions relative to the initial conditions
copyVect(RX,DX,n); //calculate direction for X
copyVect(RY,DY,n); //calculate direction for Y
copyVect(RZ,DZ,n); //calculate direction for Z

// Initializes the norms of the residue vectors and stores in
   resVectNorms
double resVectNorms[numAgents];
resVectNorms[0] = norm(RX,n);
resVectNorms[1] = norm(RY,n);
resVectNorms[2] = norm(RZ,n);

// Stop criterion
double minRes;

// Generates a seed for random numbers
srand(time(NULL));  //Generate a seed

// Count time
time_t t0, t1;

// Main loop of cooperative computation with probability P
int iterNum, shrIterNum;
t0 = clock(); // starts counting time
```

```
int numThreads, threadID;


#pragma omp parallel private(threadID,iterNum,minRes,alpha,beta,i,j
    ,k) shared(numThreads,X,Y,Z,ADX,ADY,ADZ,DX,DY,DZ,NDX,NDY,NDZ,RX,
    RY,RZ,shrIterNum)
{

  numThreads = omp_get_num_threads();
  threadID   = omp_get_thread_num();

  iterNum = 0;

  do // starts do-while loop
  {
    // Increments iteration number counter
    #pragma omp single copyprivate(iterNum)
    {
      iterNum++;
    }

    if(threadID==0){matvecsr_smm(A, I, J, DX, n, nz, ADX);}
    if(threadID==1){matvecsr_smm(A, I, J, DY, n, nz, ADY);}
    if(threadID==2){matvecsr_smm(A, I, J, DZ, n, nz, ADZ);}
    #pragma omp barrier

    // Computes the MAT matrix, needed to calculate alpha and beta
    if(threadID==0){mat11 = dotprod(DX,ADX,n);mat12 = dotprod(DX,
       ADY,n);MAT[0] = mat11;MAT[1] = mat12;MAT[3] = mat12;}
    if(threadID==1){mat13 = dotprod(DX,ADZ,n);mat22 = dotprod(DY,
       ADY,n);MAT[2] = mat13;MAT[4] = mat22;MAT[6] = mat13;}
    if(threadID==2){mat23 = dotprod(DY,ADZ,n);mat33 = dotprod(DZ,
       ADZ,n);MAT[5] = mat23;MAT[7] = mat23;MAT[8] = mat33;}
    #pragma omp barrier

    //Calculate Right-hand side to obtain matrix alpha
    if(threadID==0){MAT_b[0] = (-1)*dotprod(RX,DX,n);MAT_b[1] =
       (-1)*dotprod(RX,DY,n);MAT_b[2] = (-1)*dotprod(RX,DZ,n);}
    if(threadID==1){MAT_b[3] = (-1)*dotprod(RY,DX,n);MAT_b[4] =
       (-1)*dotprod(RY,DY,n);MAT_b[5] = (-1)*dotprod(RY,DZ,n);}
    if(threadID==2){MAT_b[6] = (-1)*dotprod(RZ,DX,n);MAT_b[7] =
       (-1)*dotprod(RZ,DY,n);MAT_b[8] = (-1)*dotprod(RZ,DZ,n);}

    #pragma omp single copyprivate(alpha)
```

```
{
  //Matrix alpha
  copyVect(MAT,MAT_original,numAgents*numAgents);
  linsys(MAT,numAgents,3,MAT_b);
  copyVect(MAT_original,MAT,numAgents*numAgents);
  for(i=0;i<numAgents*numAgents;i++)
  {
    alpha[i/numAgents][i%numAgents] = MAT_b[i];
  }
}

//Updates all agents
for(i=0;i<n;i++)
{
  if(threadID==0){X[i] = X[i] + alpha[0][0]*DX[i] + alpha
      [0][1]*DY[i] + alpha[0][2]*DZ[i];}
  if(threadID==1){Y[i] = Y[i] + alpha[1][0]*DX[i] + alpha
      [1][1]*DY[i] + alpha[1][2]*DZ[i];}
  if(threadID==2){Z[i] = Z[i] + alpha[2][0]*DX[i] + alpha
      [2][1]*DY[i] + alpha[2][2]*DZ[i];}
}

//Computes product A*X, A*Y and A*Z
if(threadID==0){matvecsr_smm(A, I, J, X, n, nz, AX);}
if(threadID==1){matvecsr_smm(A, I, J, Y, n, nz, AY);}
if(threadID==2){matvecsr_smm(A, I, J, Z, n, nz, AZ);}

//Updates all residues
for(i=0;i<n;i++)
{
  if(threadID==0){RX[i] = AX[i] - b[i];}
  if(threadID==1){RY[i] = AY[i] - b[i];}
  if(threadID==2){RZ[i] = AZ[i] - b[i];}
}



//Calculate Right-hand side to obtain matrix beta
if(threadID==0){MAT_b[0] = (-1)*dotprod(RX,ADX,n);MAT_b[1] =
    (-1)*dotprod(RX,ADY,n);MAT_b[2] = (-1)*dotprod(RX,ADZ,n);}
if(threadID==1){MAT_b[3] = (-1)*dotprod(RY,ADX,n);MAT_b[4] =
    (-1)*dotprod(RY,ADY,n);MAT_b[5] = (-1)*dotprod(RY,ADZ,n);}
if(threadID==2){MAT_b[6] = (-1)*dotprod(RZ,ADX,n);MAT_b[7] =
    (-1)*dotprod(RZ,ADY,n);MAT_b[8] = (-1)*dotprod(RZ,ADZ,n);}
```

```
    #pragma omp single copyprivate(beta)
    {
      //Matrix beta
      linsys(MAT,numAgents,3,MAT_b);
      for(i=0;i<numAgents*numAgents;i++)
      {
        beta[i/numAgents][i%numAgents] = MAT_b[i];
      }
    }

    //Updates all directions
    for(i=0;i<n;i++)
    {
      if(threadID==0){NDX[i] = RX[i] + beta[0][0]*DX[i] + beta
          [0][1]*DY[i] + beta[0][2]*DZ[i];}
      if(threadID==1){NDY[i] = RY[i] + beta[1][0]*DX[i] + beta
          [1][1]*DY[i] + beta[1][2]*DZ[i];}
      if(threadID==2){NDZ[i] = RZ[i] + beta[2][0]*DX[i] + beta
          [2][1]*DY[i] + beta[2][2]*DZ[i];}
    }

    if(threadID==0){copyVect(NDX,DX,n);}
    if(threadID==1){copyVect(NDY,DY,n);}
    if(threadID==2){copyVect(NDZ,DZ,n);}
    //#pragma omp barrier

    //Computes residues norms
    if(threadID==0){resVectNorms[0] = norm(RX,n);}
    if(threadID==1){resVectNorms[1] = norm(RY,n);}
    if(threadID==2){resVectNorms[2] = norm(RZ,n);}

    // check stop criterion
    #pragma omp single copyprivate(minRes)
    {
      minRes = minval(resVectNorms,numAgents);
      // Saves minimum residual norm
    }

  }
  while ( minRes > tol && iterNum < maxIterNum ); //end while

  shrIterNum = iterNum;

}
```

```
// Ends time count
t1 = clock();

// Open files to store residues
FILE *fminres;
char minresFN[20];
strcpy(minresFN,mmfFile);
strcat(minresFN,"mrn");
fminres= fopen(minresFN,"w");

// Open files to save the solution
solAgXfp = fopen("solAgX.txt","w");
solAgYfp = fopen("solAgY.txt","w");
solAgZfp = fopen("solAgZ.txt","w");


//Saves report file


// Date and time
time_t now;
time(&now);
fprintf(reportFP,"%s\n", ctime(&now));

fprintf(reportFP,"Execution␣report␣of␣program␣%s␣with␣3␣agents␣\n\n
    ", argv[0]);

// Input files
fprintf(reportFP,"INPUT␣FILES\n");
fprintf(reportFP,"=====␣=====\n\n");
fprintf(reportFP,"Matrix␣A␣(matrix␣market␣format):␣%s\n",mmfFile);
fprintf(reportFP,"Right-hand␣side␣vector␣b:␣%s\n",rhsFile);
fprintf(reportFP,"Initial␣condition␣vector␣for␣agent␣X:␣%s\n",
    icXFile);
fprintf(reportFP,"Initial␣condition␣vector␣for␣agent␣Y:␣%s\n",
    icYFile);
fprintf(reportFP,"Initial␣condition␣vector␣for␣agent␣Z:␣%s\n",
    icZFile);

// Input variables
fprintf(reportFP,"\nINPUT␣VARIABLES\n");
fprintf(reportFP,"=====␣=========\n\n");
fprintf(reportFP,"Agent␣X␣running␣Conjugate␣Gradient\n");
fprintf(reportFP,"Agent␣Y␣running␣Conjugate␣Gradient\n");
```

```
fprintf(reportFP,"Agent␣Z␣running␣Conjugate␣Gradient\n");

fprintf(reportFP,"Agents␣exchange␣information␣at␣each␣%d␣iterations
    \n", P);
fprintf(reportFP,"Tolerance␣(stop␣rule):␣%e\n", tol);

// Solution vectors for agent X, Y and Z
fprintf(reportFP,"\nSOLUTION␣VECTORS␣FOR␣AGENT␣X,␣Y␣AND␣Z\n");
fprintf(reportFP,"========␣=======␣===␣=====␣==========\n\n");


if ( minval(resVectNorms,3) <= tol )
{
  fprintf(reportFP,"Error:␣%e\n", minval(resVectNorms,numAgents));
  fprintf(reportFP,"Solution␣found␣in␣%d␣iterations␣within␣a␣
      tolerance␣of␣%e␣\n\n",shrIterNum,tol);
}
else
{
  fprintf(reportFP,"Algorithm␣did␣not␣satisfy␣the␣stop␣rule␣(error␣
      <=␣%e␣)␣after␣%d␣iterations.\n\n",tol, shrIterNum);
  fprintf(reportFP,"Error␣after␣%d␣iterations:␣%e\n\n",iterNum,
      minRes);
}

fprintf(reportFP,"Total␣running␣time:␣%.3f␣seconds\n",(float)((t1-
    t0)/(3*CLOCKS_PER_SEC)));

// Informs which agent found the smallest residue
int agMinRes = cblas_idamin(numAgents, resVectNorms, 1)+1;
fprintf(reportFP,"Smallest␣residue␣found␣by␣agent␣%d␣\n\n",agMinRes
    );

k=0;
for (i=0;i<n;i++)
{
  fprintf(reportFP,"x(%d)␣=␣%f␣\t||␣y(%d)␣=␣%f␣\t||␣z(%d)␣=␣%f␣\n",
      i,X[i],k,Y[i],k,Z[i]);
  k++;

  // Solutions to separate files
  fprintf(solAgXfp,"%f\n",X[i]);
  fprintf(solAgYfp,"%f\n",Y[i]);
  fprintf(solAgZfp,"%f\n",Z[i]);
}
```

```
    fprintf(reportFP,"\n----------------------------␣End␣of␣Report␣
        ----------------------------\n");

    // Closes report file
    fclose(reportFP);

    // Saves number of iterations to file a separate file
    fprintf(iterFP,"%i\n",shrIterNum);

    // Closes iterations file
    fclose(iterFP);

    // Send output to screen
    printf("\nProgram␣execution␣completed␣after␣%d␣iterations...\n",
        shrIterNum);
    printf("Total␣running␣time:␣%.3f␣seconds\n",(float)((t1-t0)/(3*
        CLOCKS_PER_SEC)));
    printf("Smallest␣residue␣found␣by␣agent␣%d␣\n",agMinRes);
    printf("\nFiles␣saved:\n");
    printf("%s␣-␣program␣execution␣report\n",report);
    printf("%s␣-␣record␣of␣number␣of␣iterations␣over␣several␣runs\n",
        iterations);
    printf("solAgX.txt,␣solAgY.txt␣and␣solAgZ.txt␣-␣solution␣computed␣
        by␣each␣agent␣X,␣Y␣and␣Z,␣respectively\n");
    printf("\n");

    ////////////////////
    // Close files
    fclose(fminres);

    // End program
    return 0;
}
```

A.3 Código do algoritmo cCG com combinação afim

```
/*
 * Program: cCG_afim.c (main routine)
 *
 * C code by Guilherme da Silva Niedu
 * Rio de Janeiro, Jun 14 2011
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "ccomputing.h" // includes function headers for cooperative
   computing
#include "mmio.h"
#include "omp.h"

int main(int argc, char *argv[])
{
    // Declare algorithm variables
    int P, F; // communicate at each P iterations
    int m, n, i, j, k; // matrix dimension and the row and column
       indices
    double T, exchangeDir; //T, exchangeDir = random numbers, P=
       probability
    double tol; // tolerance for the stop criterion

    int ret_code;
    MM_typecode matcode;


    int M, N, nz;
    int *I, *J;
    double *A;

    // Files
    FILE *f;
    FILE *rhsfp;
    FILE *icXfp;
    FILE *icYfp;
    FILE *icZfp;
    FILE *initCond; // matrix with all initial conditions
    FILE *reportFP;
    FILE *iterFP;
    FILE *solAgXfp;
```

```
FILE *solAgYfp;
FILE *solAgZfp;

// File names
char report[30];
char iterations[30];
char mmfFile[30];
char rhsFile[30];
char icXFile[30];
char icYFile[30];
char icZFile[30];

// Number of agents
int numAgents = 3; // default value, it will be overwritten if -
    numAgs is present in command line arguments
// Vector of agent values
int agentVal[numAgents];
// Maximum number of iterations
int maxIterNum;


// Check validity of the command line input arguments
if (argc < 23)
{
    fprintf(stderr, "Usage: %s -mmf [matrix-market-filename] -rhs [
        righthand side file] -icX [x0] -icY [y0] -icZ [z0] -tol [tol]
        -comm [number of iterations] -agX [agX] -agY [agY] -agZ [agZ
        ] -repfile [output-report-filename] -iterfile [file to save
        number of iterations] -maxiternum [max number of iterations]
        -numAgs [number of agents]\n", argv[0]);
    exit(1);
}

// Loads data from commandline input
for ( i=1; i<argc; i+=2 )
{

  // Matrix market file name
  if ( strcmp(argv[i],"-mmf")==0 )
  {
    f = fopen(argv[i+1],"r");
    strcpy(mmfFile,argv[i+1]);
  }

  // Right hand side file name
```

```
if ( strcmp(argv[i],"-rhs")==0 )
{
  rhsfp = fopen(argv[i+1],"r");
  strcpy(rhsFile,argv[i+1]);
}

// Initial condition for agent X
if ( strcmp(argv[i],"-icX")==0 )
{
  icXfp = fopen(argv[i+1],"r");
  strcpy(icXFile,argv[i+1]);
}

// Initial condition for agent Y
if ( strcmp(argv[i],"-icY")==0 )
{
  icYfp = fopen(argv[i+1],"r");
  strcpy(icYFile,argv[i+1]);
}

// Initial condition for agent Z
if ( strcmp(argv[i],"-icZ")==0 )
{
  icZfp = fopen(argv[i+1],"r");
  strcpy(icZFile,argv[i+1]);
}

// Probability
if ( strcmp(argv[i],"-comm")==0 )
{
  P = atoi(argv[i+1]);
}

// First Communication
if ( strcmp(argv[i],"-fcomm")==0 )
{
  F = atoi(argv[i+1]);
}

// Number of agents
if ( strcmp(argv[i],"-numAgs")==0 )
{
  numAgents = atoi(argv[i+1]);
}
```

```
  // Tolerance
  if ( strcmp(argv[i],"-tol")==0 )
  {
    tol = atof(argv[i+1]);
  }

  // Report file
  if ( strcmp(argv[i],"-repfile")==0 )
  {
    reportFP = fopen(argv[i+1],"w");
    strcpy(report,argv[i+1]);
  }

  // File with the number of iterations
  // it will contain the number of iterations over
  // several runs of the algorithm for several initial conditions
  if ( strcmp(argv[i],"-iterfile")==0 )
  {
    iterFP = fopen(argv[i+1],"a");
    strcpy(iterations,argv[i+1]);
  }

  // Specify maximum number of iterations
  if ( strcmp(argv[i],"-maxiternum")==0 )
  {
    maxIterNum = atoi(argv[i+1]);
  }

} // end for(i=1;i<argc;i+=2)

// Checks if the max number of iterations was informed
// otherwise, set it to 10000
if (maxIterNum==0)
  maxIterNum=10000;
// Loads matrix into memory in matrix market form
if (mm_read_banner(f, &matcode) != 0)
{
    printf("Could␣not␣process␣Matrix␣Market␣banner.\n");
    exit(1);
}

//  This is how one can screen matrix types if their application
//  only supports a subset of the Matrix Market data types.
if (mm_is_complex(matcode) && mm_is_matrix(matcode) &&
        mm_is_sparse(matcode) )
```

```
{
    printf("Sorry, this application does not support ");
    printf("Market Market type: [%s]\n", mm_typecode_to_str(matcode
        ));
    exit(1);
}

// find out size of sparse matrix ....
if ((ret_code = mm_read_mtx_crd_size(f, &M, &N, &nz)) !=0)
    exit(1);


// reseve memory for matrices
I = (int *) malloc(nz * sizeof(int));
J = (int *) malloc(nz * sizeof(int));
A = (double *) malloc(nz * sizeof(double)); // coefficients matrix


// NOTE: when reading in doubles, ANSI C requires the use of the "l
    "
//   specifier as in "%lg", "%lf", "%le", otherwise errors will
    occur
//  (ANSI C X3.159-1989, Sec. 4.9.6.2, p. 136 lines 13-15)


for (i=0; i<nz; i++)
{
    fscanf(f, "%d %d %lg\n", &I[i], &J[i], &A[i]);
    I[i]--;  // adjust from 1-based to 0-based
    J[i]--;
}

if (f !=stdin) fclose(f);

// Set values for m and n (rows and columns)
m = M;
n = N;


// Current state vectors
double *X = (double*)malloc(n*sizeof(double));
double *Y = (double*)malloc(n*sizeof(double));
double *Z = (double*)malloc(n*sizeof(double));

//Alpha and Beta steps
```

```
double alpha[numAgents][numAgents];
double beta[numAgents][numAgents];

// Old state vectors

// Product of matrix A by vetor x
double *AX = (double*)malloc(n*sizeof(double));
// Product of matrix A by vetor y
double *AY = (double*)malloc(n*sizeof(double));
// Product of matrix A by vetor z
double *AZ = (double*)malloc(n*sizeof(double));


// Residuals for each agent
double *RX = (double*)malloc(n*sizeof(double));
double *RY = (double*)malloc(n*sizeof(double));
double *RZ = (double*)malloc(n*sizeof(double));

// Directions for each agent
double *DX = (double*)malloc(n*sizeof(double));
double *DY = (double*)malloc(n*sizeof(double));
double *DZ = (double*)malloc(n*sizeof(double));

//New Directions for each agent
double *NDX = (double*)malloc(n*sizeof(double));
double *NDY = (double*)malloc(n*sizeof(double));
double *NDZ = (double*)malloc(n*sizeof(double));

//Matrix MAT, to compute alpha and beta
double MAT[numAgents*numAgents];
double MAT_original[numAgents*numAgents];


//Products matrix-vector to form MAT
double *ADX = (double*)malloc(n*sizeof(double));
double *ADY = (double*)malloc(n*sizeof(double));
double *ADZ = (double*)malloc(n*sizeof(double));

//Elements of MAT
double mat11, mat12, mat13, mat22, mat23, mat33;

//Right-hand side for systems involving MAT
double *MAT_b = (double*)malloc(numAgents*numAgents*sizeof(double))
    ;
```

```
//Matrix needed for communitacion step
double COMM[(numAgents-1)*(numAgents-1)];

//Elements of comm
double comm11, comm12, comm22;

//Vectors needed for communitacion step
double COMM_b[(numAgents-1)];
double *rho_1 = (double*)malloc(n*sizeof(double));
double *rho_2 = (double*)malloc(n*sizeof(double));
double *sigma_1 = (double*)malloc(n*sizeof(double));
double *sigma_2 = (double*)malloc(n*sizeof(double));
double lambda[numAgents];

// Loads initial conditions
char lineX[20];
i=0;
while(fgets(lineX,20,icXfp) != NULL)
{
     X[i] = atof(lineX);
     i++;
}
fclose(icXfp);

char lineY[20];
i=0;
while(fgets(lineY,20,icYfp) != NULL)
{
     Y[i] = atof(lineY);
     i++;
}
fclose(icYfp);

char lineZ[20];
i=0;
while(fgets(lineZ,20,icZfp) != NULL)
{
     Z[i] = atof(lineZ);
     i++;
}
fclose(icZfp);

// Loads righthand side of Ax=b
char line[20];
i=0;
```

```
double *b = (double*)malloc(n*sizeof(double));
while(fgets(line,20,rhsfp) != NULL)
{
     b[i] = atof(line);
     i++;
}
fclose(rhsfp);

//Computes product A*X, A*Y and A*Z
matvecsr_smm(A, I, J, X, n, nz, AX);
matvecsr_smm(A, I, J, Y, n, nz, AY);
matvecsr_smm(A, I, J, Z, n, nz, AZ);

//Computes the residues relative to the initial conditions
for(j=0;j<n;j++)
{
  RX[j] = AX[j]-b[j]; //calculate residue value for X
  RY[j] = AY[j]-b[j]; //calculate residue value for Y
  RZ[j] = AZ[j]-b[j]; //calculate residue value for Z
}

//Computes the directions relative to the initial conditions
copyVect(RX,DX,n); //calculate direction for X
copyVect(RY,DY,n); //calculate direction for Y
copyVect(RZ,DZ,n); //calculate direction for Z

// Initializes the norms of the residue vectors and stores in
   resVectNorms
double resVectNorms[numAgents];
resVectNorms[0] = norm(RX,n);
resVectNorms[1] = norm(RY,n);
resVectNorms[2] = norm(RZ,n);

// Stop criterion
double minRes;

// Generates a seed for random numbers
srand(time(NULL));  //Generate a seed

// Count time
time_t t0, t1;

// Main loop of cooperative computation with probability P
int iterNum, shrIterNum;
t0 = clock(); // starts counting time
```

```
int numThreads, threadID;


#pragma omp parallel private(threadID,iterNum,minRes,alpha,beta,i,j
    ,k) shared(numThreads,X,Y,Z,ADX,ADY,ADZ,DX,DY,DZ,NDX,NDY,NDZ,RX,
    RY,RZ,shrIterNum)
{

  numThreads = omp_get_num_threads();
  threadID   = omp_get_thread_num();

  iterNum = 0;

  do // starts do-while loop
  {
    // Increments iteration number counter
    #pragma omp single copyprivate(iterNum)
    {
      iterNum++;
    }

    if(threadID==0){matvecsr_smm(A, I, J, DX, n, nz, ADX);}
    if(threadID==1){matvecsr_smm(A, I, J, DY, n, nz, ADY);}
    if(threadID==2){matvecsr_smm(A, I, J, DZ, n, nz, ADZ);}
    #pragma omp barrier

    // Computes the MAT matrix, needed to calculate alpha and beta
    if(threadID==0){mat11 = dotprod(DX,ADX,n);mat12 = dotprod(DX,
       ADY,n);MAT[0] = mat11;MAT[1] = mat12;MAT[3] = mat12;}
    if(threadID==1){mat13 = dotprod(DX,ADZ,n);mat22 = dotprod(DY,
       ADY,n);MAT[2] = mat13;MAT[4] = mat22;MAT[6] = mat13;}
    if(threadID==2){mat23 = dotprod(DY,ADZ,n);mat33 = dotprod(DZ,
       ADZ,n);MAT[5] = mat23;MAT[7] = mat23;MAT[8] = mat33;}
    #pragma omp barrier

    //Calculate Right-hand side to obtain matrix alpha
    if(threadID==0){MAT_b[0] = (-1)*dotprod(RX,DX,n);MAT_b[1] =
       (-1)*dotprod(RX,DY,n);MAT_b[2] = (-1)*dotprod(RX,DZ,n);}
    if(threadID==1){MAT_b[3] = (-1)*dotprod(RY,DX,n);MAT_b[4] =
       (-1)*dotprod(RY,DY,n);MAT_b[5] = (-1)*dotprod(RY,DZ,n);}
    if(threadID==2){MAT_b[6] = (-1)*dotprod(RZ,DX,n);MAT_b[7] =
       (-1)*dotprod(RZ,DY,n);MAT_b[8] = (-1)*dotprod(RZ,DZ,n);}

    #pragma omp single copyprivate(alpha)
```

```
{
  //Matrix alpha
  copyVect(MAT,MAT_original,numAgents*numAgents);
  linsys(MAT,numAgents,3,MAT_b);
  copyVect(MAT_original,MAT,numAgents*numAgents);
  for(i=0;i<numAgents*numAgents;i++)
  {
    alpha[i/numAgents][i%numAgents] = MAT_b[i];
  }
}

//Updates all agents
for(i=0;i<n;i++)
{
  if(threadID==0){X[i] = X[i] + alpha[0][0]*DX[i] + alpha
      [0][1]*DY[i] + alpha[0][2]*DZ[i];}
  if(threadID==1){Y[i] = Y[i] + alpha[1][0]*DX[i] + alpha
      [1][1]*DY[i] + alpha[1][2]*DZ[i];}
  if(threadID==2){Z[i] = Z[i] + alpha[2][0]*DX[i] + alpha
      [2][1]*DY[i] + alpha[2][2]*DZ[i];}
}

//Computes product A*X, A*Y and A*Z
if(threadID==0){matvecsr_smm(A, I, J, X, n, nz, AX);}
if(threadID==1){matvecsr_smm(A, I, J, Y, n, nz, AY);}
if(threadID==2){matvecsr_smm(A, I, J, Z, n, nz, AZ);}

//Updates all residues
for(i=0;i<n;i++)
{
  if(threadID==0){RX[i] = AX[i] - b[i];}
  if(threadID==1){RY[i] = AY[i] - b[i];}
  if(threadID==2){RZ[i] = AZ[i] - b[i];}
}
#pragma omp barrier

#pragma omp single
{
  //Coomunication step
  if(P>0)
  {
    if((iterNum-F)%P==0)
    {
      for(i=0;i<n;i++)
      {
```

```
  rho_1[i] = RX[i] - RZ[i];
  rho_2[i] = RY[i] - RZ[i];
  sigma_1[i] = X[i] - Z[i];
  sigma_2[i] = Y[i] - Z[i];
}
comm11 = dotprod(rho_1,sigma_1,n);
comm12 = dotprod(rho_1,sigma_2,n);
comm22 = dotprod(rho_2,sigma_2,n);
COMM[0] = comm11; COMM[1] = comm12;
COMM[2] = comm12; COMM[3] = comm22;
COMM_b[0] = (-1)*dotprod(RZ,sigma_1,n);
COMM_b[1] = (-1)*dotprod(RZ,sigma_2,n);
linsys(COMM,numAgents-1,1,COMM_b);

lambda[0] = COMM_b[0]; lambda[1] = COMM_b[1];
lambda[2] = 1.0 - COMM_b[0] - COMM_b[1];


if((iterNum/P)%numAgents==1)
{
  for(i=0;i<n;i++)
  {
    RX[i] = lambda[0]*RX[i] + lambda[1]*RY[i] + lambda
        [2]*RZ[i];
    X[i] = lambda[0]*X[i] + lambda[1]*Y[i] + lambda[2]*Z[
        i];
  }
}
if((iterNum/P)%numAgents==2)
{
  for(i=0;i<n;i++)
  {
    RY[i] = lambda[0]*RX[i] + lambda[1]*RY[i] + lambda
        [2]*RZ[i];
    Y[i] = lambda[0]*X[i] + lambda[1]*Y[i] + lambda[2]*Z[
        i];
  }
}
if((iterNum/P)%numAgents==0)
{
  for(i=0;i<n;i++)
  {
    RZ[i] = lambda[0]*RX[i] + lambda[1]*RY[i] + lambda
        [2]*RZ[i];
    Z[i] = lambda[0]*X[i] + lambda[1]*Y[i] + lambda[2]*Z[
```

```
                i];
          }
        }

      }
    }
}



//Calculate Right-hand side to obtain matrix beta
if(threadID==0){MAT_b[0] = (-1)*dotprod(RX,ADX,n);MAT_b[1] =
    (-1)*dotprod(RX,ADY,n);MAT_b[2] = (-1)*dotprod(RX,ADZ,n);}
if(threadID==1){MAT_b[3] = (-1)*dotprod(RY,ADX,n);MAT_b[4] =
    (-1)*dotprod(RY,ADY,n);MAT_b[5] = (-1)*dotprod(RY,ADZ,n);}
if(threadID==2){MAT_b[6] = (-1)*dotprod(RZ,ADX,n);MAT_b[7] =
    (-1)*dotprod(RZ,ADY,n);MAT_b[8] = (-1)*dotprod(RZ,ADZ,n);}


#pragma omp single copyprivate(beta)
{
  //Matrix beta
  linsys(MAT,numAgents,3,MAT_b);
  for(i=0;i<numAgents*numAgents;i++)
  {
    beta[i/numAgents][i%numAgents] = MAT_b[i];
  }
}

//Updates all directions
for(i=0;i<n;i++)
{
  if(threadID==0){NDX[i] = RX[i] + beta[0][0]*DX[i] + beta
      [0][1]*DY[i] + beta[0][2]*DZ[i];}
  if(threadID==1){NDY[i] = RY[i] + beta[1][0]*DX[i] + beta
      [1][1]*DY[i] + beta[1][2]*DZ[i];}
  if(threadID==2){NDZ[i] = RZ[i] + beta[2][0]*DX[i] + beta
      [2][1]*DY[i] + beta[2][2]*DZ[i];}
}

if(threadID==0){copyVect(NDX,DX,n);}
if(threadID==1){copyVect(NDY,DY,n);}
if(threadID==2){copyVect(NDZ,DZ,n);}

//Computes residues norms
```

```
    if(threadID==0){resVectNorms[0] = norm(RX,n);}
    if(threadID==1){resVectNorms[1] = norm(RY,n);}
    if(threadID==2){resVectNorms[2] = norm(RZ,n);}

    // check stop criterion
    #pragma omp single copyprivate(minRes)
    {
      minRes = minval(resVectNorms,numAgents);
      // Saves minimum residual norm
     }

  }
  while ( minRes > tol && iterNum < maxIterNum ); //end while

  shrIterNum = iterNum;

}

// Ends time count
t1 = clock();

// Open files to store residues
FILE *fminres;
char minresFN[20];
strcpy(minresFN,mmfFile);
strcat(minresFN,"mrn");
fminres= fopen(minresFN,"w");

// Open files to save the solution
solAgXfp = fopen("solAgX.txt","w");
solAgYfp = fopen("solAgY.txt","w");
solAgZfp = fopen("solAgZ.txt","w");


//Saves report file


// Date and time
time_t now;
time(&now);
fprintf(reportFP,"%s\n", ctime(&now));

fprintf(reportFP,"Execution␣report␣of␣program␣%s␣with␣3␣agents␣\n\n
   ", argv[0]);
```

```
// Input files
fprintf(reportFP,"INPUT␣FILES\n");
fprintf(reportFP,"=====␣=====\n\n");
fprintf(reportFP,"Matrix␣A␣(matrix␣market␣format):␣%s\n",mmfFile);
fprintf(reportFP,"Right-hand␣side␣vector␣b:␣%s\n",rhsFile);
fprintf(reportFP,"Initial␣condition␣vector␣for␣agent␣X:␣%s\n",
    icXFile);
fprintf(reportFP,"Initial␣condition␣vector␣for␣agent␣Y:␣%s\n",
    icYFile);
fprintf(reportFP,"Initial␣condition␣vector␣for␣agent␣Z:␣%s\n",
    icZFile);

// Input variables
fprintf(reportFP,"\nINPUT␣VARIABLES\n");
fprintf(reportFP,"=====␣=========\n\n");
fprintf(reportFP,"Agent␣X␣running␣Conjugate␣Gradient\n");
fprintf(reportFP,"Agent␣Y␣running␣Conjugate␣Gradient\n");
fprintf(reportFP,"Agent␣Z␣running␣Conjugate␣Gradient\n");

fprintf(reportFP,"Agents␣exchange␣information␣at␣each␣%d␣iterations
    \n", P);
fprintf(reportFP,"Tolerance␣(stop␣rule):␣%e\n", tol);

// Solution vectors for agent X, Y and Z
fprintf(reportFP,"\nSOLUTION␣VECTORS␣FOR␣AGENT␣X,␣Y␣AND␣Z\n");
fprintf(reportFP,"========␣=======␣===␣=====␣==========\n\n");


if ( minval(resVectNorms,3) <= tol )
{
  fprintf(reportFP,"Error:␣%e\n", minval(resVectNorms,numAgents));
  fprintf(reportFP,"Solution␣found␣in␣%d␣iterations␣within␣a␣
      tolerance␣of␣%e␣\n\n",shrIterNum,tol);
}
else
{
  fprintf(reportFP,"Algorithm␣did␣not␣satisfy␣the␣stop␣rule␣(error␣
      <=␣%e␣)␣after␣%d␣iterations.\n\n",tol, shrIterNum);
  fprintf(reportFP,"Error␣after␣%d␣iterations:␣%e\n\n",iterNum,
      minRes);
}

fprintf(reportFP,"Total␣running␣time:␣%.3f␣seconds\n",(float)((t1-
    t0)/(3*CLOCKS_PER_SEC)));
```

```
// Informs which agent found the smallest residue
int agMinRes = cblas_idamin(numAgents, resVectNorms, 1)+1;
fprintf(reportFP,"Smallest residue found by agent %d \n\n",agMinRes
    );

k=0;
for (i=0;i<n;i++)
{
  fprintf(reportFP,"x(%d) = %f \t || y(%d) = %f \t || z(%d) = %f \n",
      i,X[i],k,Y[i],k,Z[i]);
  k++;

  // Solutions to separate files
  fprintf(solAgXfp,"%f\n",X[i]);
  fprintf(solAgYfp,"%f\n",Y[i]);
  fprintf(solAgZfp,"%f\n",Z[i]);
}
fprintf(reportFP,"\n----------------------------- End of Report
    -----------------------------\n");

// Closes report file
fclose(reportFP);

// Saves number of iterations to file a separate file
fprintf(iterFP,"%i\n",shrIterNum);

// Closes iterations file
fclose(iterFP);

// Send output to screen
printf("\nProgram execution completed after %d iterations...\n",
    shrIterNum);
printf("Total running time: %.3f seconds\n",(float)((t1-t0)/(3*
    CLOCKS_PER_SEC)));
printf("Smallest residue found by agent %d \n",agMinRes);
printf("\nFiles saved:\n");
printf("%s - program execution report\n",report);
printf("%s - record of number of iterations over several runs\n",
    iterations);
printf("solAgX.txt, solAgY.txt and solAgZ.txt - solution computed
    by each agent X, Y and Z, respectively\n");
printf("\n");

//////////////////
// Close files
```

```
    fclose(fminres);

    // End program
    return 0;
}
```

A.4 Código para gerar matrizes Positivas Definidas

```
/*
 * Program: Gerador.c (main routine)
 *
 * C code by Guilherme da Silva Niedu
 * Rio de Janeiro, Jun 14 2011
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "ccomputing.h" // includes function headers for cooperative
   computing
#include "mmio.h"

double aleat()
{
    int q = 1000;
    int r = rand();
    int a = r%(2*q+1);
    a-=q;
    return ((double)a)/((double)q);
}

int main()
{
    srand(time(NULL));
    int N = 11000;
    double *U = (double*)malloc(N*N*sizeof(double));
    double *v = (double*)malloc(N*sizeof(double));
    double *v_temp = (double*)malloc(N*sizeof(double));
    double *p_temp = (double*)malloc(N*sizeof(double));
    int i, j, k, l;
    double w, nrm;

    for(i = 0;i<N;i++)
    {
        v[i] = aleat();
    }
    nrm = norm(v,N);
    for(i = 0;i<N;i++)
    {
        U[i] = v[i]/nrm;
    }
```

```
for(i=1;i<N;i++)
{
    for(j = 0;j<N;j++)
    {
        v[j] = aleat(); //Generates random column
    }
    copyVect(v,v_temp,N); //copies it, to deduct(qi'*v)*qi from v
    for(k = 0; k<i; k++)
    {
        w = dotprod(v,&U[k*N],N); //computes qi'*v
        copyVect(&U[k*N],p_temp,N);
        for(l = 0; l<N; l++)
        {
            p_temp[l] = p_temp[l] * w; //multiplies qi by qi'*v
        }
        for(l = 0; l<N; l++)
        {
            v_temp[l] = v_temp[l] - p_temp[l]; //deducts (qi'*v)*qi
                from v
        }
    }
    nrm = norm(v_temp,N);
    for(j = 0;j<N;j++)
    {
        U[i*N + j] = v_temp[j]/nrm;
    }
}
double *L = (double*)malloc(N*sizeof(double));
int cond = 1e6; //condition number
L[0] = 1;
L[N-1] = cond;
for(i=1;i<N-1;i++)
{
    L[i] = (rand())%(cond+1);
                            if(L[i]==0)
                            {
                                    L[i]==1000;
                            }
}
            //Orthogonal matrix generated
double *LUt = (double*)malloc(N*N*sizeof(double));
for(i=0;i<N;i++)
{
    for(j=0;j<N;j++)
    {
```

```
            LUt[i*N + j] = U[i*N + j] * L[i];
        }
    }
    //Lambda*U computed
    double *A = (double*)malloc(N*N*sizeof(double));
    A = matprod(U, N, N, "N", LUt, N, N, "T");
    //Matrix A computed
    int aa, bb;
    FILE *file;
    file = fopen("Matriz11000.mtx","w");
    fprintf(file,"%%%%MatrixMarket␣matrix␣coordinate␣real␣symmetric\n")
       ;
    fprintf(file,"%d␣%d␣%d\n", N, N, N*(N+1)/2);
    for(bb=0;bb<N;bb++)
    {
        for(aa=bb;aa<N;aa++)
        {
            fprintf(file,"%d␣%d␣%.13e", aa+1, bb+1, A[aa*N + bb]);
            if(bb!=N-1 || aa!=N-1){fprintf(file,"\n");}
        }
    }
    fclose(file);
    FILE *file2;
    file2 = fopen("Matriz11000_AV.eig","w");
    FILE *file_b, *file_x, *file_y, *file_z;
    file_b = fopen("Matriz11000_b.rhs","w");
    file_x = fopen("Matriz11000_x0.ic","w");
    file_y = fopen("Matriz11000_y0.ic","w");
    file_z = fopen("Matriz11000_z0.ic","w");
    for(aa=0;aa<N;aa++)
    {
        fprintf(file2,"%.6e\n",L[aa]);
        fprintf(file_b,"%.6f\n",aleat());
        fprintf(file_x,"%.6f\n",aleat());
        fprintf(file_y,"%.6f\n",aleat());
        fprintf(file_z,"%.6f\n",aleat());
    }
    fclose(file2);
    fclose(file_b);
    fclose(file_x);
    fclose(file_y);
    fclose(file_z);
    //Matrix A , right-hand side b and initial conditions written to
       file
    return 0;
```

```
}
```