Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Interoperabilidade entre ALGENCAN e Julia

Autor: Maicon Lucas Mares de Souza
Orientador: Prof. Dr. John Lenon Cardoso Gardenghi

Brasília, DF
2023

Maicon Lucas Mares de Souza

Interoperabilidade entre ALGENCAN e Julia

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. John Lenon Cardoso Gardenghi

Brasília, DF

2023

Maicon Lucas Mares de Souza
Interoperabilidade entre ALGENCAN e Julia/ Maicon Lucas Mares de Souza. – Brasília, DF, 2023-
72 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. John Lenon Cardoso Gardenghi

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2023.

1. Interoperabilidade. 2. ALGENCAN. I. Prof. Dr. John Lenon Cardoso Gardenghi. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Interoperabilidade entre ALGENCAN e Julia

CDU 02:141:005.6

Maicon Lucas Mares de Souza

Interoperabilidade entre ALGENCAN e Julia

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 14 de dezembro de 2023 – Data da aprovação do trabalho:

Prof. Dr. John Lenon Cardoso Gardenghi
Orientador

Prof. Dr. Bruno César Ribas
Convidado 1

Prof. Dr. Glauco Vitor Pedrosa
Convidado 2

Brasília, DF
2023

Dedico este trabalho primeiramente a minha mãe, que sempre me deu todo o suporte e me apoiou em todas as minhas decisões, e como minha contribuição ao desenvolvimento da humanidade.

"Be ashamed to die until you have won some victory for humanity".

Horace Mann

Resumo

Diversas são as aplicações de otimização no dia a dia, como a minimização de recursos e maximização de lucros em produções industriais, diminuição do tráfego terrestre em grandes metrópoles, montar refeição com todos os nutrientes necessários e o mais barato possível para crianças em uma escola, entre outros. Entretanto, o número de variáveis e restrições presentes nesses problemas podem crescer rapidamente tornando impraticável a resolução manual. Com o objetivo de reduzir o trabalho manual do usuário e fornecer resultados confiáveis para tais problemas foi desenvolvido o *solver Augmented Lagrangian with GENCAN* (ALGENCAN) no trabalho realizado pelos professores R. Andreani, E. G. Birgin, J. M. Martínez, e M. L. Schuverdt, em 2008, e melhorado posteriormente, em 2020. Este *solver* é escrito em Fortran, uma linguagem estruturada e fortemente tipada. Nos últimos anos, foram propostas algumas linguagens de cunho científico matemático, como alternativas às linguagens clássicas como Fortran e C. Dentre elas, destaca-se a linguagem Julia, uma linguagem que traz aspectos produtivos devido à alta abstração. A linguagem Julia permite a flexibilidade na declaração de tipos, sendo possível utilizá-la com os tipos declarados explicitamente ou deixar que os tipos sejam definidos pelo compilador em tempo de compilação. O presente trabalho se propôs a implementar uma interoperabilidade entre a linguagem Julia e o *solver* ALGENCAN. Para realizá-lo foram utilizados os recursos da linguagem Julia que fornecem suporte às chamadas a funções em Fortran.

Palavras-chave: Interoperabilidade. Otimização não linear. ALGENCAN.

Abstract

There are a lot of applications of optimization in day-to-day, like resources minimization and profits maximization in industrial productions, diminishing terrestrial traffic on large cities, make a meal with all needed ingredients and as cheaper as possible for children at a school, among others. However, the number of variables and constraints present in that can grow fastly turning inviable a manual solving. In order to turn those problems resolution praticable and less slow was developed the Augmented Lagrangian with GENCAN library in the work conducted by the professors s R. Andreani, E. G. Birgin, J. M. Martínez, and M. L. Schuverdt, in 2008, and improved later, in 2020. That solver is written in Fortran, an structured language that requires significant number of lines of code to describe an optimization problem. In the last years, were proposed some scientific mathematical languages as alternatives to classic languages like Fortran and C. Among those, Julia language stands out, a language that brings productive aspects due to high abstraction. In the same time Julia language brings productivity, it looks for keep good performance aspects that are present in those classic languages. The current work has proposed to implement an interoperability between the Julia language and the ALGENCAN solver. To accomplish that the resources from Julia language that offer support to Fortran function calls were applied.

Key-words: Interoperability. Nonlinear optimization. ALGENCAN.

Lista de ilustrações

Figura 1 – KNITRO visão geral 18
Figura 2 – Exemplo de matriz esparsa 19
Figura 3 – Exemplo formato COO 20
Figura 4 – Exemplo formato CSR 20
Figura 5 – Compilação do código fonte em Fortran 25
Figura 6 – Criação da biblioteca compartilhada 25
Figura 7 – Listar nomes das funções 26
Figura 8 – Chamando código Fortran a partir de Julia 26
Figura 9 – Descompactação do pacote ALGENCAN 31
Figura 10 – Definição da variável de ambiente 32
Figura 11 – Compilação da biblioteca BLAS 32
Figura 12 – Compilação da biblioteca HSL 33
Figura 13 – Compilação da biblioteca ALGENCAN 33
Figura 14 – Mapa das cidades e antena 34
Figura 15 – Solução do problema pelo ALGENCAN 38
Figura 16 – O modelo cascata 39
Figura 17 – Cascata modificado 41
Figura 18 – Fluxo de execução da interface em Julia 45
Figura 19 – Compilação da biblioteca compartilhada 49
Figura 20 – Compilação da biblioteca compartilhada 49
Figura 21 – Execução da interface em Julia 49
Figura 22 – Execução dos testes em Julia 51
Figura 23 – Execução dos testes em Fortran 52
Figura 24 – Variação no tempo de execução entre ALGENCAN via Julia e ALGENCAN via Fortran 55

Lista de tabelas

Tabela 1 – Parâmetros para a sub-rotina evalf 28
Tabela 2 – Parâmetros para a sub-rotina evalg 28
Tabela 3 – Parâmetros para a sub-rotina evalc 29
Tabela 4 – Parâmetros para a sub-rotina evalj 30
Tabela 5 – Parâmetros para a sub-rotina evalhl 31
Tabela 6 – Localização do centro das cidades 35
Tabela 7 – Requisitos Funcionais 44
Tabela 8 – Requisitos Não Funcionais 44
Tabela 9 – Comparação do tempo de execução para problemas rodados com ALGENCAN nativamente em Fortran e usando a interface em Julia. Na tabela, $f(x^*)$ representa o valor final da função objetivo e é considerado apenas para mostrar que ambas convergiram marginalmente à mesma solução. 54

Sumário

1 INTRODUÇÃO 12
1.1 Revisão Bibliográfica 12
1.2 Problema 13
1.3 Objetivos 14
1.4 Metodologia 14
1.5 Estrutura do Documento 15

2 REFERENCIAL TEÓRICO 16
2.1 Programação não linear 16
2.2 Trabalhos Correlatos 17
2.2.1 KNITRO 17
2.2.2 IPOPT 18
2.3 Matrizes esparsas 19
2.3.1 Formato COO 19
2.3.2 Formato CSR 20
2.4 Linguagem Julia 21
2.4.1 Tipagem opcional de variáveis 21
2.4.2 *Multiple Dispatch* 21
2.4.3 Compilador JIT 22
2.4.4 Tipos Paramétricos 22
2.4.5 Interoperabilidade com Fortran 23

3 ALGENCAN 27
3.1 Definição 27
3.2 Estrutura 27
3.2.1 evalf 28
3.2.2 evalg 28
3.2.3 evalc 29
3.2.4 evalj 29
3.2.5 evalhl 29
3.3 Configuração e Instalação 30
3.3.1 Compilação das dependências 31
3.3.2 Compilação ALGENCAN 33
3.3.3 Configuração automatizada 33
3.4 Exemplo de aplicação 34

4 METODOLOGIA 39
4.0.1 Seleção do processo para o desenvolvimento da interface 40
4.0.2 Emprego do Processo Cascata no desenvolvimento da interface 40
4.1 Fase de Testes 41

5 IMPLEMENTAÇÃO DA INTERFACE 43
5.1 *Design* da Interface 44
5.2 Implementação 45
5.2.1 Utilização da Interface 48
5.3 Testes 49
5.4 Documentação 52

6 ANÁLISE DOS RESULTADOS 53

7 CONCLUSÃO 56

REFERÊNCIAS 57

APÊNDICES 61

APÊNDICE A – INTERFACE COM O CUTEST EM JULIA 62

APÊNDICE B – SCRIPT PARA GUIAR OS TESTES EM JULIA . 71

1 Introdução

A Otimização é uma ferramenta fundamental na tomada de decisão em problemas cujo sistema de equações analisado envolve múltiplas variáveis. Para entendê-la, é necessário identificar uma função que caracteriza o comportamento do sistema sendo analisado, conhecida comumente por função objetivo (MITTAL, 2015).

A função objetivo representa uma quantidade ou uma combinação de quantidades, que pode ser, uma medida custo, lucro, desempenho ou outra quantidade de interesse. Tal função depende de características do sistema em análise, chamadas de variáveis. As variáveis podem estar restringidas por meio de equações e desigualdades, chamadas de restrições. Ao fazer uso da otimização o objetivo está em encontrar os valores das variáveis que otimizam a função objetivo (NOCEDAL; WRIGHT, 2006). Se ao menos uma dentre a função objetivo e as restrições for uma equação não linear, tem-se um problema de otimização não linear com restrições (MITTAL, 2015).

Após formulado o problema, algoritmos de otimização com o auxílio de um computador são comumente utilizados para a resolução. Não existe um algoritmo de otimização que sozinho resolva todos os problemas de otimização, mas sim uma coleção de algoritmos para problemas específicos (VENTER, 2010). A escolha deve ser definida pelo usuário.

Quanto aos algoritmos para a resolução de problemas de otimização não linear com restrições, existem diversas opções. O Método de Lagrangiano Aumentado é uma delas, sendo um algoritmo bastante conhecido para a resolução dos problemas desse tipo (KANZOW; RAHARJA; SCHWARTZ, 2021).

1.1 Revisão Bibliográfica

Diante dos problemas de otimização não linear com restrições sobre os quais há interesse neste trabalho, o *solver* ALGENCAN é uma opção de auxílio ao usuário (ANDREANI et al., 2008). O mesmo implementa o Método de Lagrangiano Aumentado e é escrito em Fortran. Para descrever o problema a ser resolvido e fazer uso do *solver*, é exigido do usuário a codificação de algumas sub-rotinas.

A codificação das sub-rotinas devem respeitar regras em suas implementações, tipos de dados e parâmetros requeridos, além das características próprias da linguagem Fortran (BIRGIN; MARTíNEZ, 2014).

Similar ao ALGENCAN, existem outras opções para a mesma classe de problemas, porém implementados em outras linguagens, como o Nlopt, originalmente escrito em C++ (JOHNSON, 2011), o Nlsolve, desenvolvido totalmente em Julia (MOGENSEN; RISETH,

2018), o Ipopt, implementado em C++ (WäCHTER; BIEGLER, 2006) e o Knitro 5.0, escrito em linguagem C (BYRD; NOCEDAL; WALTZ, 2006).

Observa-se o emprego de algumas linguagens dinâmicas que facilitam o desenvolvimento de interoperabilidade para os *solvers*, entre elas a linguagem Julia. Julia é uma linguagem de programação dinâmica de alto desempenho que vem crescendo rapidamente nos últimos anos (BEZANSON et al., 2014). Foi desenvolvida como um projeto de código aberto que teve início em 2009 e sua primeira versão lançada em 2012 (BEZANSON et al., 2012). Desde então, mais de 2 milhões de usuários têm acessado o *site* oficial da linguagem adotando-a como ferramenta de ensino em universidades do mundo todo (BEZANSON et al., 2014). A comunidade abrange mais de 500 contribuidores e contribuiu com cerca de 1200 pacotes (BEZANSON et al., 2014).

Julia oferece *solvers* para a resolução de problemas de otimização, bem como suporte, por meio de seu recurso de interoperabilidade, para uso de *solvers* já conhecidos pela comunidade (LUBIN; DUNNING, 2015).

A linguagem Julia não possui a necessidade de ser compilada manualmente para código de máquina como Fortran e C. Há um compilador embutido, *Just In Time* (JIT), que permite compilar código desenvolvido em Julia para código de máquina em tempo de execução, o que torna possível que ela alcance desempenho similar ao de linguagens compiladas (RANOCHA et al., 2022).

Ainda é possível escrever código com sintaxe muito próxima à linguagem matemática. O recurso de interoperabilidade presente na linguagem Julia é um ponto interessante que cabe destaque. Por meio desse recurso, é possível o suporte a chamadas a funções em outras linguagens nativamente, sendo viável fazer chamadas a procedimentos escritos em Fortran a partir de Julia por exemplo (BEZANSON et al., 2012).

Podemos definir o conceito de interoperabilidade como a habilidade de dois ou mais sistemas trabalharem em conjunto (MALONE, 2014). Essa definição é ampla e cobre desde grupos de pessoas a pedaços de *hardware*. Aqui sistema é utilizado para se referir às diferentes linguagens de programação. O principal objetivo da interoperabilidade entre linguagens de programação é combinar as suas diferentes capacidades, pois cada linguagem desempenha determinado papel melhor em determinadas situações (MALONE, 2014).

1.2 Problema

O ALGENCAN é um projeto conhecido no meio científico e possui características vantajosas, como os formatos para armazenamento e operação com matrizes esparsas e o processo de aceleração para a convergência da solução,que o diferenciam dos demais *solvers* para a mesma classe de problemas (BIRGIN; MARTíNEZ, 2014). Contudo, os

usuários estão restritos a codificar somente na linguagem Fortran se desejarem fazer uso do *solver* ALGENCAN.

Dada a popularidade da linguagem Julia, a variedade de *solvers* que são usados em Julia (vários dos quais escritos em outra linguagem de programação) e o recurso de interoperabilidade para a linguagem Fortran, observa-se uma oportunidade de expandir o uso do *solver* ALGENCAN.

A pergunta de pesquisa é: *é viável do ponto de vista implementação e tempo de execução ofertar o uso do ALGENCAN na linguagem Julia?*

1.3 Objetivos

O objetivo deste trabalho foi desenvolver uma interoperabilidade entre a Julia e o ALGENCAN. A esta interoperabilidade deu-se o nome de interface.

Para alcançar o Objetivo Geral serão adotados os objetivos específicos:

- Definir um processo de desenvolvimento de *software* adequado ao contexto do trabalho.
- Projetar a estrutura da interface de modo a respeitar as restrições e comportamento do *solver* ALGENCAN.
- Validar a interface desenvolvida.

1.4 Metodologia

Para a implementação da interface foi escolhido o modelo de processo cascata. O processo foi modificado para se adequar ao contexto específico do trabalho. No processo Cascata Modificado foram executadas as fases: Levantamento de Requisitos, *Design* da interface, Implementação, Testes e Documentação.

A validação da interface desenvolvida foi realizada na Fase de Testes do processo Cascata Modificado. Para a validação foi escolhido o pacote *a Constrained and Unconstrained Testing Environment with safe threads for mathematical optimization* (CUTEst) (GOULD; ORBAN; TOINT, 2015). Foram selecionados problemas de otimização não linear a partir do pacote *CUTEst*.

Os dados coletados nos testes envolveram a mensuração do tempo necessário para a resolução dos problemas e a sua correta resolução. Foram mensurados o tempo utilizando somente o *solver* ALGENCAN e o tempo necessário utilizando a interface que faz chamada ao ALGENCAN. Os resultados finais nos dois cenários foram comparados em tabela.

1.5 Estrutura do Documento

Este trabalho está divido da seguinte maneira: no Capítulo 2 é apresentado todo o Referencial Teórico necessário para o entendimento do trabalho, que inclui os trabalhos correlatos, os métodos matemáticos e formatos de matriz implementados no ALGENCAN, características da linguagem Julia, funcionamento da interoperabilidade entre Julia e Fortran e a biblioteca de testes utilizada para a validação do desenvolvimento.

O Capítulo 3 é dedicado ao entendimento do ALGENCAN, desde sua instalação e configuração até a sua execução.

O Capítulo 4 apresenta a Metodologia, que indica como cada um dos objetivos serão alcançados.

O Capítulo 5 aborda a execução do processo de desenvolvimento adotado que traz detalhes da interface desenvolvida e sua estrutura.

O Capítulo 6 traz a análise dos resultados do trabalho desenvolvido.

O Capítulo 7 traz a Conclusão deste trabalho a partir do Objetivo Geral, da pergunta de pesquisa e dos resultados colhidos, bem como as recomendações de trabalhos futuros.

2 Referencial Teórico

Este capítulo traz os conceitos necessários para o entendimento do presente trabalho. São abordados os conceitos de programação não linear, trabalhos correlatos, isto é, exemplos de *solvers* para problemas não lineares e que existe interface em Julia para seu uso, matrizes esparsas, e os formatos adotados pelo ALGENCAN para seu processamento, e uma visão geral da linguagem Julia.

2.1 Programação não linear

Programação não linear é um campo da otimização que consiste em um conjunto de técnicas, métodos e teoremas para a resolução de problemas não lineares de otimização (ANTONIOU, 2021). Nesses problemas a função objetivo e ou, geralmente, as restrições são não lineares (ANTONIOU, 2021).

Problemas de programação não linear possuem a seguinte configuração:

$$\begin{aligned} \min\,(\text{ou max})\ & f(x) \\ \text{s. a}\ h_j(x) &= 0 \quad \text{para cada } j \in \{1,...,m\}, \\ g_i(x) &\leq 0 \quad \text{para cada } i \in \{1,...,p\}, \\ \text{para}\ \ x &\geq 0, \end{aligned} \tag{2.1}$$

f, h_j e g_i, para $j \in \{1,\ldots,m\}$ e $i \in \{1,\ldots,p\}$ são funções escalares, com $x \in R^n$ um vetor de n elementos. Alguns algoritmos de programação não linear exigem que as funções sejam todas continuamente diferenciáveis, isto é, cuja derivada exista em cada ponto do domínio dessas funções.

O Método do Lagrangiano é um método utilizado em programação não linear e tem como objetivo solucionar problemas de otimização como o problema 2.1. Esse método resolve o problema por meio da função Lagrangiana, tal função relaciona a função objetivo com as restrições (SANTOS, 2011). A função Lagrangiana é definida por

$$L(x,\lambda,\mu) = f(x) + \sum_{i=1}^{m} \lambda_i h_i(x) + \sum_{i=1}^{p} \mu_i g_i(x), \tag{2.2}$$

sendo $f(x)$ a função objetivo do problema, $h(x)$ o vetor de restrições de igualdade e $g(x)$ o vetor de restrições de desigualdade, e λ e μ os multiplicadores de Lagrange para as restrições de igualdade e desigualdade, respectivamente.

O Método do Lagrangiano Aumentado é uma combinação dos métodos do Lagrangiano e da Penalidade (SANTOS, 2011). Nesse método, a função Lagrangiana recebe um parâmetro de penalidade, esse termo é responsável por penalizar as restrições e guiar

a otimização em direção de soluções viáveis (ANDREANI et al., 2007). Dessa maneira, é relacionada a função objetivo e as restrições com as penalidades aplicadas a essas. A função Lagrangiana aumentada é definida por:

$$L(x, \lambda, \mu, \rho) = f(x) + \frac{\rho}{2}[\sum_{i=1}^{m}(h_i(x) + \frac{\lambda_i}{\rho})^2 + \sum_{i=1}^{p}(g_i(x) + \frac{\mu_i}{\rho})^2], \tag{2.3}$$

onde ρ é o fator de penalidade, as demais funções e parâmetros são os mesmos já abordados na equação 2.2.

O Método do Lagrangiano Aumentado, traz em sua implementação o método dos multiplicadores que é uma variação do método da Penalidade e possui a vantagem do termo de penalidade não necessitar tender ao infinito para que a convergência seja garantida e assim alivia a instabilidade numérica (MASIERO, 2011). Uma das desvantagens é a necessidade de se escolher inicialmente o parâmetro de penalidade e os multiplicadores de Lagrange (MASIERO, 2011).

2.2 Trabalhos Correlatos

Nesta seção, o objetivo é descrever sucintamente dois *solvers* clássicos da literatura, similares ao ALGENCAN, para a resolução de problemas de programação não linear, e que também possuem interoperabilidade com Julia, apesar de serem escritos em outras linguagens.

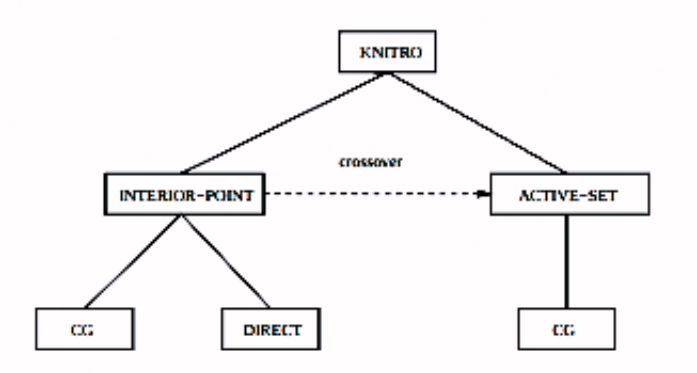
2.2.1 KNITRO

Nonlinear Interior point Trust Region Optimization (KNITRO) é um pacote desenvolvido em linguagem C para otimização não linear que mescla diferentes técnicas complementares entre si para resolver problemas de larga escala, entretanto não se limita apenas a problemas de otimização não lineares (BYRD; NOCEDAL; WALTZ, 2006). As técnicas que o algoritmo usa são: método de pontos interiores e método de restrições ativas. É implementado dois algoritmos com abordagens diferentes para a técnica método de pontos interiores e um algoritmo para o método de restrições ativas. Ainda são fornecidas técnicas de mesclagem (*crossover*) que permitem combinar ambas as técnicas citadas em determinadas situações.

A técnica método de pontos interiores trabalha com uma sequência de aproximações para o problema original. Um conjunto de subproblemas são projetados para convergir para a solução ótima do problema e são mais fáceis de serem resolvidos se comparados ao problema original. Uma versão dessa técnica utiliza uma abordagem de gradiente conjugado para calcular o passo do algoritmo, enquanto uma segunda versão calcula o passo por meio de uma fatorização direta do sistema linear em questão (BYRD; NOCEDAL; WALTZ, 2006).

Por outro lado, o método de restrições ativas encontra a solução ótima para o problema por meio da resolução de subproblemas lineares. Em cada passo, é identificado um subconjunto das restrições, o conjunto ativo, que afeta a solução ótima, então a técnica segue utilizando somente essas restrições para resolver o subproblema. Diferente da técnica descrita anteriormente, o método de restrições ativas utiliza somente fatorização direta para os sistemas lineares (BYRD; NOCEDAL; WALTZ, 2006). Uma visualização melhor dos algoritmos é apresentada na imagem a seguir:

Figura 1 – KNITRO visão geral



Fonte: (BYRD; NOCEDAL; WALTZ, 2006)

Os problemas não lineares a serem resolvidos possuem o seguinte formato:

$$\begin{aligned} & f(x) \\ \text{s. a } & C_E(x) = 0 \\ & C_I(x) \leq 0, \end{aligned} \tag{2.4}$$

onde $f : R^n \to R, C_E : R^n \to R^l$ e $C_I : R^n \to R^m$, ambas são funções continuamente diferenciáveis.

2.2.2 IPOPT

Interior Point Optimizer (IPOPT) é um *solver* escrito em C++, que assim como o explicado anteriormente, também focado na resolução de problemas não lineares e baseado na técnica método de pontos interiores. O *software* é livre e pode ser baixado diretamente a partir de seu repositório original [1]. Esta técnica utiliza uma função obstáculo para transformar o problema não linear original em uma sequência de problemas aproximados e estes são resolvidos por algum método análogo ao de Newton, no caso específico do IPOPT, é utilizado o método de Newton esparso aplicado a condições *Karush–Kuhn–Tucker* (KKT)

[1] https://github.com/coin-or/Ipopt

(BIEGLER; ZAVALA, 2009). A função obstáculo cria uma barreira para a região não viável do problema, em consequência o algoritmo é guiado para o interior da região viável. O algoritmo inicia com um ponto viável, mas longe da solução original, e move para mais próximo da solução ótima a cada passo por meio da redução do parâmetro de obstáculo utilizado na função obstáculo (BIEGLER; ZAVALA, 2009).

2.3 Matrizes esparsas

Matrizes esparsas são encontradas com frequência nos cálculos envolvidos nos mais diversos tipos de projetos científicos. Qualquer matriz em que a fração entre o número de elementos não nulos e o total de elementos da matriz, é baixa, é considerada uma matriz esparsa (MOHAMMED; MEHMOOD, 2022). Um exemplo de matriz esparsa pode ser visto na Figura 2.

Figura 2 – Exemplo de matriz esparsa

$$\begin{bmatrix} 0 & -1 & 0 & 0 & 9 \\ 0 & 0 & 2 & 0 & 0 \\ 7 & 0 & 3 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 \end{bmatrix}$$

Fonte: Autor

As matrizes esparsas que aparecem em problemas do dia a dia são grandes, mas são compostas em sua maioria, por elementos nulos (MOHAMMED; MEHMOOD, 2022). Devido à quantidade de elementos nulos, o desempenho das operações sobre tais matrizes pode ser diretamente afetado. Portanto requerem formatos e algoritmos especializados para o seu acesso, cálculo e armazenamento (MOHAMMED; MEHMOOD, 2022).

Neste trabalho foram utilizados os formatos de armazenamento *Compressed Sparse Row* (CSR) e *Coordinate List* (COO) que visam o armazenamento eficiente de matrizes esparsas.

2.3.1 Formato COO

É um esquema de armazenamento relativamente simples. Uma matriz esparsa armazenada nesse formato consiste em 3 vetores (DANG; SCHMIDT, 2012):

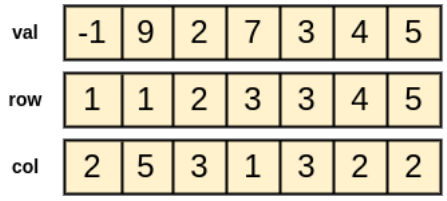
- *val*: armazena os valores não nulos da matriz original.
- *row*: armazena os índices das linhas dos valores não nulos.

- *col*: armazena os índices das colunas dos valores não nulos.

Cada um desses vetores terá dimensão N_z, sendo N_z a quantidade de elementos não nulos presentes na matriz esparsa.

O formato COO aplicado na matriz da Figura 2 é representado na Figura 3.

Figura 3 – Exemplo formato COO



Fonte: Autor

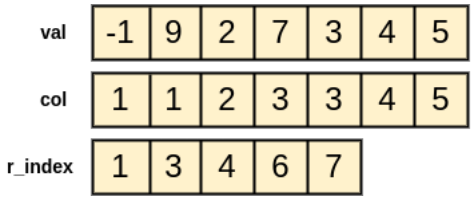
2.3.2 Formato CSR

O formato CSR é o formato mais comum e um dos primeiros utilizado para armazenar matrizes esparsas (ALAHMADI et al., 2020). O formato se baseia em 3 vetores para armazenar a matriz esparsa:

- *val*: contém os elementos não nulos da matriz esparsa.
- *col*: é utilizado para armazenar os índices das colunas dos elementos não nulos.
- *r_index*: essa matriz tem como objetivo o armazenamento dos índices dos elementos em *val* que são os primeiros elementos não nulos de suas respectivas linhas na matriz esparsa

A matriz esparsa da Figura 2 quando aplicado o formato CSR pode ser representada conforme a Figura 4.

Figura 4 – Exemplo formato CSR



Fonte: Autor

2.4 Linguagem Julia

Julia é uma linguagem dinâmica de propósito geral, que suporta diversos paradigmas de programação, como imperativo, funcional e de orientação a objetos, que estende e adapta diversas técnicas de linguagens modernas para fornecer desempenho ao mesmo tempo que fornece produtividade. O *design* da linguagem é baseado em funções genéricas e em um rico sistema de tipos que permite inferência de tipos (BEZANSON et al., 2012).

Diversas características presentes no núcleo dessa linguagem são responsáveis pelo seu alto desempenho e sua escolha para este trabalho, entre elas podemos citar:

- Tipagem opcional de variáveis;
- *Multiple Dispatch*;
- Compilador JIT;
- Tipos paramétricos;
- Interoperabilidade com Fortran.

2.4.1 Tipagem opcional de variáveis

Um dos recursos que contribui para o alto desempenho de linguagens de programação é um sistema de tipos estático, já que o mesmo evita o tempo gasto com inferência de tipos durante o tempo de execução (KEMMER; RJASANOW; HILDEBRANDT, 2018).

A tipagem opcional possibilita que o usuário utilize Julia como uma linguagem de alto nível ou uma linguagem verbosa de baixo nível com os tipos declarados explicitamente e ainda no mesmo projeto (KEMMER; RJASANOW; HILDEBRANDT, 2018). Como consequência, partes críticas de um projeto que necessitam de alto desempenho podem ser implementadas com tipagem explícita e assim evitar a inferência de tipos, enquanto operações triviais podem optar pelo uso de tipagem implícita.

No presente trabalho será feito o uso de tipagem explícita em todos os casos, tanto na definição de variáveis e também na de funções com seus argumentos. Na chamada das funções definidas dessa forma, essa escolha resulta em funções que conservam a precisão de argumentos de ponto flutuante e não fazem conversões de tipos. Isso permite que se evite a inferência de tipos e a interface desenvolvida seja consistente.

2.4.2 *Multiple Dispatch*

Multiple dispatch é uma funcionalidade da Linguagem Julia que permite definir funções genéricas com diversos métodos associados a essas, cada um desses para diferentes combinações de argumentos da função (JULIALANG, 2022b). Dessa maneira, são

possíveis diferentes implementações da mesma função e a implementação a ser escolhida dependerá dos argumentos fornecidos à função em tempo de execução (KEMMER; RJASANOW; HILDEBRANDT, 2018).

Para o presente trabalho se torna viável, através do recurso *Multiple Dispatch*, a implementação de interfaces uniformes (KEMMER; RJASANOW; HILDEBRANDT, 2018). Pois não são necessárias verificações condicionais, verificação de tipos ou outras indireções, já que o método a ser especializado para lidar com os tipos de argumentos passados é definido em tempo de execução (BEZANSON et al., 2012).

2.4.3 Compilador JIT

O compilador JIT é responsável por transformar código de alto nível em código de máquina nativo em tempo de execução (VERDUGO; BADIA, 2022). O código gerado é especializado para os tipos encontrados em tempo de execução e, portanto, possui desempenho próximo ao de linguagens de alto desempenho como C, C++ e Fortran (VERDUGO; BADIA, 2022).

A fim de compensar o tempo adicional para recompilar pacotes importados e código anteriormente já compilado, Julia possibilita que esses sejam compilados somente uma vez e reutilizados através de *cache* (KEMMER; RJASANOW; HILDEBRANDT, 2018). Além desse recurso, diversas rotinas nativas de Julia são *wrappers* para outras funções externas cujas bibliotecas são escritas em outras linguagens e assim se beneficiam da otimização já presente nas mesmas (KEMMER; RJASANOW; HILDEBRANDT, 2018).

Essa funcionalidade presente em Julia, é nada mais do que uma biblioteca Lower Level Virtual Machine (LLVM) escrita em C++ responsável pela geração de código de máquina. Os caminhos e passos pelos quais um trecho de código passa podem variar (JULIALANG, 2022b).

No geral, o código passa por um *parser* que o transforma em uma representação da forma *Abstract Syntax Tree* (AST), tal forma transforma o código em *tokens*, pois estes são mais adequados para manipulação e execução. Em seguida, a representação AST é convertida em instruções LLVM que, por sua vez, são otimizadas e transformadas em código *Assembly* nativo (JULIALANG, 2022b).

2.4.4 Tipos Paramétricos

Os tipos paramétricos são tipos que podem ser definidos por meio de um ou mais parâmetros, tais parâmetros podem ser utilizados para definir o comportamento ou propriedades do tipo em questão e consequentemente habilitam a verificação estática de tipos (JULIALANG, 2022c).

Essa funcionalidade permite ao sistema de tipos da Linguagem Julia inferir os tipos em tempo de compilação e o compilador pode rastrear os tipos dos valores até mesmo quando esses são armazenados em estruturas de dados mutáveis (BEZANSON et al., 2012). Existem múltiplas razões do alto desempenho devido ao uso de tipos paramétricos como, por exemplo, reuso de código, implementação de algoritmos genéricos, polimorfismo e outros.

2.4.5 Interoperabilidade com Fortran

A linguagem Julia oferece suporte nativo, através da interface *ccall*, à chamada de funções desenvolvidas em C ou Fortran e, portanto, não requer código escrito em outras linguagens de programação para que seja feita a comunicação (LOBIANCO, 2019).

A interface permite carregar as bibliotecas e chamar as suas funções dinamicamente, isto é, sem a necessidade de uma compilação prévia ou vincular a biblioteca estaticamente (JULIALANG, 2022a). A chamada às funções é direta, não há necessidade de geração de código, um código de cola padrão ou compilação, mesmo estando no *prompt* interativo (JULIALANG, 2022a).

Para invocar uma função ou sub-rotina desenvolvida em Fortran é necessário utilizar o macro *@ccall* ou a palavra reservada *ccall*. Outro requisito é que a função a ser chamada faça parte de uma biblioteca compartilhada (JULIALANG, 2022a).

A fim de exemplificar o uso da interface *ccall* será criada uma biblioteca compartilhada a partir do Código 1, desenvolvido em Fortran, ao qual terá a sua função *compute_media* invocada através do macro *@ccall* no Código 2.

A função *computed_media*, definida no Código 1, recebe 3 parâmetros, sendo eles e nesta ordem: uma função do tipo *callback* definida em Julia e que será invocada dentro do código em Fortran e os dois últimos são valores de ponto flutuante.

Ao utilizar o macro *@ccall* para chamar uma função é necessário seguir a sintaxe do Código 3.

Os identificadores presentes no Código 3 significam (JULIALANG, 2022a):

1. *library*: corresponde à biblioteca alvo, ao qual deve ser uma *string* constante ou literal indicando o caminho da biblioteca. A biblioteca pode ser omitida caso faça parte do processo corrente.
2. *argtypes*: os tipos de entrada que correspondem à assinatura da função.
3. *argvalues*: os valores que de fato serão passados para a função.
4. *returntype*: O tipo de retorno da função.

```
module fortran_client
  use iso_fortran_env, only: dp => real64
  implicit none

  abstract interface
    function sum_callback(a, b) result(res)
      import dp
      real(dp), intent(in) :: a, b
      real(dp) :: res
    end function sum_callback
  end interface

contains
  function compute_media(f, a, b) result(res)
    real(dp), intent(in) :: a, b
    procedure(sum_callback) :: f
    real(dp) :: res

    print *, "Received a: ", a
    print *, "Received b: ", b

    ! Call Julia function to compute res = a + b
    res = f(a, b)/2
  end function compute_media
end module fortran_client
```

Código 1: Código a ser chamado em Fortran

```
function sum_callback(x::Float64, y::Float64)::Float64
  return x + y
end

function main()
  a::Float64 = 10.0
  b::Float64 = 20.0
  sum_callback_ptr = @cfunction(sum_callback, Float64, (Ref{Float64}, Ref{Float64}))
  res = @ccall "./libfortran_client.so".__fortran_client_MOD_compute_media(
    sum_callback_ptr::Ptr{Cvoid},
    a::Ref{Float64}, b::Ref{Float64}
  )::Float64
  println(res)
end
```

Código 2: Uso da interface ccall em Julia

```
1       @ccall library.function_name(argvalue1::argtype1, ...)::returntype
```

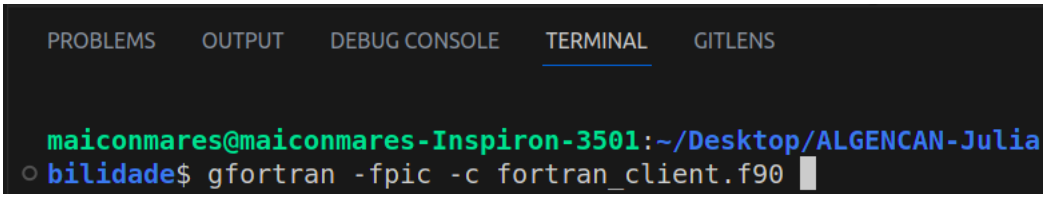
Código 3: Syntaxe do macro @ccall

Os tipos dos argumentos passados (*argtypes*) para a interface *ccall* são avaliados estaticamente e, portanto, não podem haver referências a variáveis locais (JULIALANG, 2022a). Se todos os argumentos corresponderem à assinatura da função implementada, a chamada será realizada, caso contrário, um erro será levantado.

A partir da especificação da assinatura da função a ser chamada, os valores passados na interface (*argvalues*) e o valor retornado por ela são convertidos para os tipos correspondentes na linguagem alvo por meio de uma chamada feita à função *convert* em Julia, logo não há preocupação por parte do usuário para lidar com conversões de tipos (BEZANSON et al., 2012).

Uma vez tendo implementado o código em Fortran, é necessário realizar sua compilação e gerar uma biblioteca compartilhada contendo o mesmo. Optou-se neste trabalho por utilizar o compilador Gfortran, ao qual faz parte da suíte GNU Compiler Collection (GCC), no processo de compilação (STALLMAN, 2009). Na Figura 5 é compilado o código fonte que irá gerar um arquivo de mesmo nome, mas com extensão ".o", e que contém o código objeto.

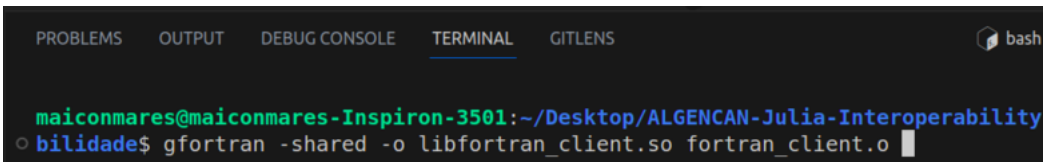
Figura 5 – Compilação do código fonte em Fortran



Fonte: Autor

Na Figura 6 é criada a biblioteca compartilhada a partir do código objeto gerado no passo anterior.

Figura 6 – Criação da biblioteca compartilhada



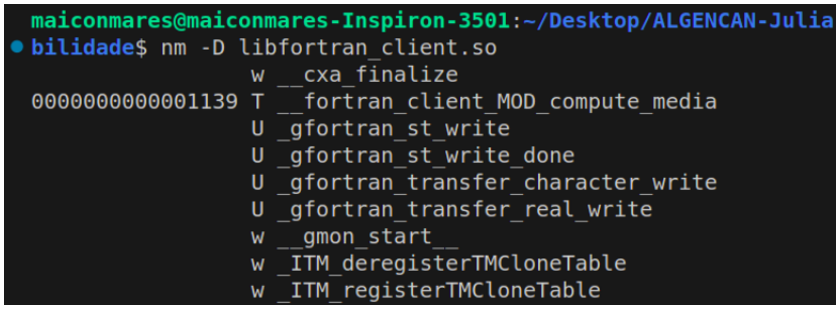
Fonte: Autor

Após a criação da biblioteca, a função desejada pode então ser invocada. Porém,

um problema comum enfrentado ao utilizar compiladores Fortran, são os *mangled names*. Compiladores Fortran por padrão seguem regras que afetam os nomes das funções, modificando-as para caixa alta ou caixa baixa e ou adicionando *underscores*, tal processo é conhecido como *name mangling* (JULIALANG, 2022a).

Para visualizar os nomes das funções presentes em uma biblioteca compartilhada após a sua criação pode ser utilizada a ferramenta nm do GCC conforme a Figura 7 (STALLMAN, 2009).

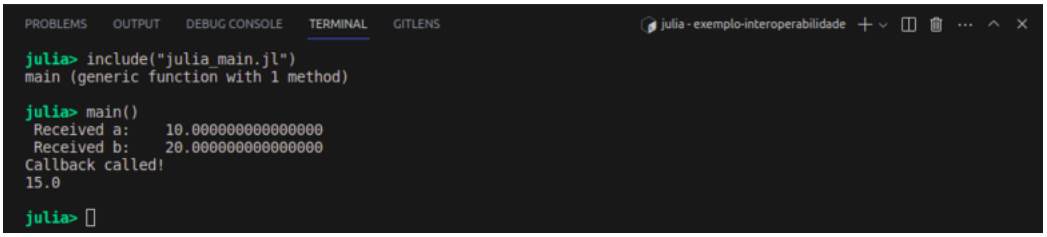
Figura 7 – Listar nomes das funções



Fonte: Autor

Em seguida, é possível chamar a função desejada e obter o resultado como pode ser visto na figura 8.

Figura 8 – Chamando código Fortran a partir de Julia



Fonte: Autor

3 ALGENCAN

3.1 Definição

ALGENCAN é implementado como uma sub-rotina em Fortran, ao qual chamamos de *solver* neste trabalho, que utiliza técnicas de Lagrange Aumentado para resolver problemas de otimização restringidos. Essa sub-rotina faz parte de um conjunto de sub-rotinas para otimização desenvolvidas no Projeto *Trustable Algorithms for Nonlinear General Optimization* (TANGO) [1] no Departamento de Matemática Aplicada da Universidade Estadual de Campinas e no Departamento de Ciência da Computação da Universidade de São Paulo, sendo um *software* brasileiro.

O *solver* foi desenvolvido no trabalho de Birgin e Martínez (2014) e, mais tarde, implementada uma nova versão, 4.0.0, melhorada no trabalho de Birgin e Martínez (2020). Esta última versão é a que iremos trabalhar no presente trabalho.

Os problemas resolvidos por ALGENCAN são da forma:

$$\min_{x \in R^n} f(x) \text{ sujeito a } h(x) = 0,\ g(x) \leq 0, \text{ e } l \leq x \leq u, \tag{3.1}$$

onde $f : R^n \to R, h : R^n \to R^m$, e $g : R^n \to R^p$ são funções continuamente diferenciáveis, sendo n a dimensão do vetor x, m define a dimensão do vetor $h(x)$ de restrições de igualdade e, por fim, p define a dimensão do vetor $g(x)$ de restrições de desigualdade. Por fim, l e u são vetores de restrição de limites inferior e superior impostas aos valores do vetor x, respectivamente.

O algoritmo começa por adicionar um termo de penalidade à função objetivo, que tem como propósito desencorajar a violação das restrições do problema. Pontos candidatos, que se espera que satisfaçam às restrições, são calculados e utilizados para construir um modelo da função objetivo. A cada iteração o termo de penalidade, os multiplicadores de Lagrange e os pontos candidatos são atualizados até um critério de parada ser satisfeito (BIRGIN; MARTíNEZ, 2014).

3.2 Estrutura

Esta seção tem como objetivo explicar o funcionamento e utilização do *solver* ALGENCAN.

[1] https://www.ime.usp.br/ egbirgin/tango/

Ao utilizar o ALGENCAN, é necessário que especifiquemos alguns parâmetros iniciais que descrevem o problema, bem como é exigido que o usuário codifique 5 sub-rotinas, em Fortran, que calculam partes do problema necessárias para a resolução do problema como um todo (BIRGIN; MARTíNEZ, 2014).

3.2.1 evalf

Dada a função objetivo, $f(x)$, que descreve o problema, a sub-rotina *evalf* tem como propósito o cálculo dessa função. Os parâmetros de entrada e saída e seus tipos são descritos na Tabela 1.

Tabela 1 – Parâmetros para a sub-rotina evalf

parâmetro	descrição	tipo	entrada	saída
n	dimensão do vetor x	integer	sim	não
x	vetor com os valores de x	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	sim	não
f	armazena o resultado da função objetivo	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	não	sim
inform	armazena códigos de erro, caso algum erro ocorra	integer	sim	sim
pdataptr	tipo composto que armazena a quantidade de chamadas às rotinas de avaliação, sendo customizável pelo usuário (opcional)	c_ptr	sim	não

Fonte: Autor

3.2.2 evalg

A rotina *evalg*, por sua vez, tem como objetivo o cálculo do gradiente da função objetivo. O gradiente da função objetivo, representado como $\nabla f(x)$, será uma matriz que conterá todas as derivadas parciais de primeira ordem da função objetivo (BIRGIN; MARTíNEZ, 2014). Tal matriz possui ordem $n \times 1$, sendo que n indica o total de variáveis da função objetivo.

Os parâmetros de entrada e saída para a sub-rotina *evalg* são descritos na Tabela 2.

Tabela 2 – Parâmetros para a sub-rotina evalg

parâmetro	descrição	tipo	entrada	saída
n	dimensão do vetor x	integer	sim	não
x	vetor com os valores de x	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	sim	não
g	armazena o resultado do gradiente da função objetivo	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	não	sim
inform	armazena códigos de erro, caso algum erro ocorra	integer	sim	sim
pdataptr	tipo composto que armazena a quantidade de chamadas às rotinas de avaliação, sendo customizável pelo usuário (opcional)	c_ptr	sim	não

Fonte: Autor

3.2.3 evalc

A terceira sub-rotina, *evalc*, realiza o cálculo das restrições do problema. São calculadas as $m+p$ restrições e armazenadas em um vetor denotado por c, sendo m restrições de igualdade e p restrições de desigualdade. No vetor que armazena as restrições computadas, primeiro estão dispostas as m restrições e em seguida as p restrições.

Os parâmetros para a sub-rotina *evalc* são listados na Tabela 3.

Tabela 3 – Parâmetros para a sub-rotina evalc

parâmetro	descrição	tipo	entrada	saída
n	dimensão do vetor x	integer	sim	não
x	vetor com os valores de x	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	sim	não
m	quantidade de restrições de igualdade	integer	sim	não
p	quantidade de restrições de desigualdade	integer	sim	não
c	vetor que contém o resultado das m+p restrições	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	não	sim
inform	armazena códigos de erro, caso algum erro ocorra	integer	sim	sim
pdataptr	tipo composto que armazena a quantidade de chamadas às rotinas de avaliação, sendo customizável pelo usuário (opcional)	c_ptr	sim	não

Fonte: Autor

3.2.4 evalj

A sub-rotina *evalj* tem como finalidade computar a Matriz Jacobiana das restrições. Tal matriz armazena, em cada uma de suas linhas, o gradiente transposto de cada uma das restrições. Ou seja, cada linha irá conter todas as derivadas parciais de primeira ordem de uma determinada restrição. Sendo uma matriz de ordem $(m + p) \times n$, de tal forma que a modelagem no programa explora a esparsidade da matriz.

O ALGENCAN utiliza o formato CSR para armazenar a Matriz Jacobiana das restrições. A sua disposição é representada a seguir (3.2).

$$J = \begin{pmatrix} \nabla h_1(x)^T \\ \vdots \\ \nabla h_m(x)^T \\ \vdots \\ \nabla g_p(x)^T \end{pmatrix}_{(m+p)\times n} \tag{3.2}$$

Os parâmetros de entrada e saída da sub-rotina *evalj* são apresentados na Tabela 4.

3.2.5 evalhl

A última sub-rotina, *evalhl*, é responsável por computar a Hessiana da função Lagrangiana. A matriz Hessiana de uma função de várias variáveis é uma matriz que

Tabela 4 – Parâmetros para a sub-rotina evalj

parâmetro	descrição	tipo	entrada	saída
n	dimensão do vetor x	integer	sim	não
x	vetor com os valores de x	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	sim	não
m	quantidade de restrições de igualdade	integer	sim	não
p	quantidade de restrições de desigualdade	integer	sim	não
ind	vetor de m+p posições que indica quais restrições devem ser computadas na Matriz Jacobiana	logical	sim	não
sorted	vetor de m+p posições que indica se os indíces das variáveis em jvar estão em ordem crescente	logical	não	sim
jsta	vetor de m+p posições que armazena os índices dos elementos em jval que são os primeiros elementos não nulos de suas linhas na Matriz Jacobiana	integer	não	sim
jlen	vetor de m+p posições que armazena em cada índice a quantidade de elementos não nulos da respectiva linha na Matriz Jacobiana	integer	não	sim
lim	define o número máximo de entradas em jvar e jval	integer	sim	não
jvar	vetor que armazena em cada índice o número da coluna do respectivo elemento em jval	integer	não	sim
jval	vetor que armazena os elementos não nulos da Matriz Jacobiana	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	não	sim
inform	armazena códigos de erro, caso algum erro ocorra	integer	sim	sim
pdataptr	tipo composto que armazena a quantidade de chamadas às rotinas de avaliação, sendo customizável pelo usuário (opcional)	c_ptr	sim	não

Fonte: Autor

contém todas as derivadas parciais da função. Considerando que a função Lagrangiana é dada por:

$$L(x, \lambda, \mu) = f(x) + \sum_{i=1}^{m} \lambda_i h_i(x) + \sum_{i=1}^{p} \mu_j g_j(x),$$

sua Hessiana é dada por

$$\nabla^2 L(x, \lambda, \mu) = \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 h_i(x) + \sum_{i=1}^{p} \mu_j \nabla^2 g_j(x), \tag{3.3}$$

sendo $\nabla^2 f(x)$, $\nabla^2 h_i(x)$, $i \in \{1, 2, ..., m\}$, e $\nabla^2 g_j(x)$, $j \in \{1, 2, ..., p\}$, as matrizes hessianas das funções $f(x)$, $h_i(x)$ e $g_j(x)$ que compõem a Matriz Hessiana $\nabla^2 L(x, \lambda, \mu)$. Cada uma dessas matrizes irá conter as derivadas parciais de segunda ordem de suas respectivas funções. As variáveis λ_i e μ_j representam os multiplicadores de Lagrange. Os multiplicadores de Lagrange são calculados e passados pelo ALGENCAN à função *evalhl* a cada passo.

A Matriz Hessiana do Lagrangiano é armazenada por meio do formato COO. Os seus parâmetros podem ser vistos na Tabela 5.

3.3 Configuração e Instalação

Para a instalação do ALGENCAN, um dos componentes do Projeto TANGO, é necessário realizar o seu *download* no site oficial [2], instalar algumas bibliotecas e definir variáveis de ambiente.

O Projeto TANGO está sob a licença GNU *General Public License* (GPL), assim todos os seus componentes são *software* livre e podem ser redistribuídos e modificados debaixo dos termos da GNU GPL.

[2] https://www.ime.usp.br/ egbirgin/tango/downloads.phpform

Tabela 5 – Parâmetros para a sub-rotina evalhl

parâmetro	descrição	tipo	entrada	saída
n	dimensão do vetor x	integer	sim	não
x	vetor com os valores de x	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	sim	não
m	quantidade de restrições de igualdade	integer	sim	não
p	quantidade de restrições de desigualdade	integer	sim	não
lambda	vetor de m+p posições que indica quais restrições devem ser computadas na Matriz Jacobiana	logical	sim	não
lim	define o número máximo de entradas em hlcol, hlrow e hlval	integer	sim	não
inclf	indica se a Hessiana da função objetivo deve ser incluída ou não na Matriz Hessiana da Função Lagrangiana	logical	sim	não
hlnnz	indica a quantidade de elementos não nulos armazenados em hlval	integer	não	sim
hlrow	vetor que armazena o número das linhas da matriz original da Função Lagrangiana de cada elemento em hlval	integer	não	sim
hlcol	vetor que armazena o número das colunas da matriz original da Função Lagrangiana de cada elemento em hlval	integer	não	sim
hlval	vetor que armazena os elementos não nulos da matriz original da Função Lagrangiana	real(kind=8) (ponto flutuante de 8 bytes, equivalente a Float64)	não	sim
inform	armazena códigos de erro, caso algum erro ocorra	integer	sim	sim
pdataptr	tipo composto que armazena a quantidade de chamadas às rotinas de avaliação, sendo customizável pelo usuário (opcional)	c_ptr	sim	não

Fonte: Autor

A demonstração de instalação aqui realizada é feita no sistema operacional Ubuntu 20.04.6 *Long Term Support* (LTS).

Após baixado o arquivo compactado, descompacte-o em um diretório separado. A descompactação realizada pode ser vista na Figura 9.

Figura 9 – Descompactação do pacote ALGENCAN

```
maiconmares@maiconmares-Inspiron-3501:~/Desktop$ tar -xvf algencan-4.0.0.tar.xz
algencan-4.0.0/
algencan-4.0.0/algencan-master/
algencan-4.0.0/algencan-master/.gitignore
algencan-4.0.0/algencan-master/Makefile
algencan-4.0.0/algencan-master/README
algencan-4.0.0/algencan-master/VERSION
algencan-4.0.0/algencan-master/make-dist
algencan-4.0.0/algencan-master/sources/
algencan-4.0.0/algencan-master/sources/algencan/
```

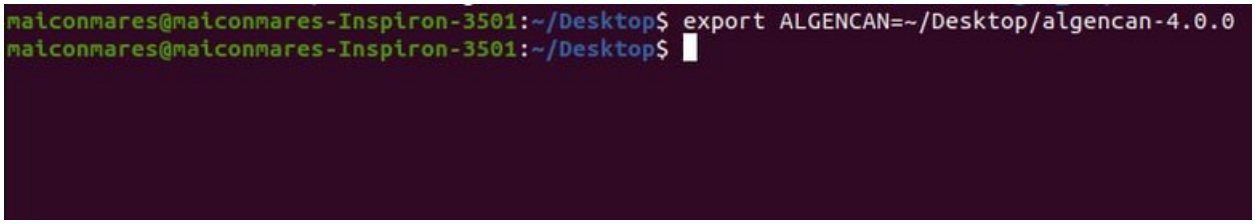
Fonte: Autor

É necessário definir em uma variável de ambiente o caminho onde o arquivo baixado foi descompactado. Por meio de tal variável a configuração do *solver* será facilitada devido a essa indicar o diretório de localização diretamente nos passos seguintes. A definição da variável deve ser feita conforme a Figura 10.

3.3.1 Compilação das dependências

O ALGENCAN faz uso de bibliotecas já conhecidas no meio científico para realizar parte de seu trabalho. Essas bibliotecas são: *Basic Linear Algebra Subprograms* (BLAS) e *Harwell Subroutine Library* (HSL).

Figura 10 – Definição da variável de ambiente



Fonte: Autor

A biblioteca BLAS define rotinas de baixo nível que realizam operações de álgebra linear muito utilizadas e foi desenvolvida em Fortan (LAWSON et al., 1979). A biblioteca pode ser baixada no site oficial [3]. O subconjunto de arquivos a serem baixados e requeridos pelo ALGENCAN são: *dgemm.f, dgemv.f, dtpmv.f, dtpsv.f, idamax.f, lsame.f* e *xerbla.f*.

É necessário que todos os arquivos baixados da biblioteca BLAS sejam movidos para a pasta **$ALGENCAN/sources/blas**. Em seguida, os arquivos precisam ser compilados e a biblioteca gerada deve ser movida para o diretório correto conforme a Figura 11.

Figura 11 – Compilação da biblioteca BLAS

Fonte: Autor

Por outro lado, a biblioteca HSL é um conjunto de pacotes desenvolvidos em Fortran para computação científica de larga escala (HOGG; REID; SCOTT, 2011). Os arquivos que compõem a HSL e requeridos pelo ALGENCAN são: *fakemetis.f, hsl_zd11d.f90, mc21ad.f, mc47ad.f, mc64ad.f, hsl_ma57d.f90, ma57ad.f, mc34ad.f, mc59ad.f* e *mc71ad.f*. Todos os arquivos podem ser baixados no site oficial [4].

É necessário que todos os arquivos baixados da HSL sejam movidos para a pasta **$ALGENCAN/sources/hsl**. Em seguida, os arquivos precisam ser compilados e a biblioteca gerada deve ser movida para o diretório correto conforme a Figura 12.

[3] https://www.netlib.org/blas/
[4] https://www.hsl.rl.ac.uk/catalogue/ma57.html

Figura 12 – Compilação da biblioteca HSL

```
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/hsl_zd11d.f90
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/hsl_ma57d.f90
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/ma57ad.f
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/mc34ad.f
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/mc47ad.f
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/mc59ad.f
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/mc64ad.f
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/mc21ad.f
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/mc71ad.f
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 $ALGENCAN/sources/hsl/fakemetis.f
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ ar rcs libhsl.a hsl_zd11d.o hsl_ma57d.o ma57ad.o mc34ad.o mc47ad.o \
> mc59ad.o mc64ad.o mc21ad.o mc71ad.o fakemetis.o
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ mv libhsl.a $ALGENCAN/sources/hsl/lib
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ mv hsl_ma57_double.mod $ALGENCAN/sources/hsl/inc/
```

Fonte: Autor

3.3.2 Compilação ALGENCAN

Após todas as bibliotecas necessárias estarem compiladas e em seus respectivos diretórios, podemos então compilar a biblioteca ALGENCAN.

Para compilar os arquivos que compõem a biblioteca e mover a biblioteca gerada e os módulos necessários para os diretórios esperados deve-se executar os comandos conforme a Figura 13.

Figura 13 – Compilação da biblioteca ALGENCAN

```
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$ gfortran -c -O3 -Wall -I$ALGENCAN/sources/hsl/inc $ALGENCAN/sources/algencan/lss
.f90 \
> gfortran -c -O3 -Wall $ALGENCAN/sources/algencan/gencan.f90 \
> gfortran -c -O3 -Wall $ALGENCAN/sources/algencan/memev.f90 \
> gfortran -c -O3 -Wall $ALGENCAN/sources/algencan/feasgencan.f90 \
> gfortran -c -O3 -Wall $ALGENCAN/sources/algencan/newtkkt.f90 \
> gfortran -c -O3 -Wall $ALGENCAN/sources/algencan/algencan.f90 \
> ar rcs libalgencan.a lss.o gencan.o memev.o feasgencan.o newtkkt.o algencan.o \
> mv libalgencan.a $ALGENCAN/sources/algencan/lib \
> mv bmalgencan.mod $ALGENCAN/sources/algencan/inc \
> mv bmgencan.mod $ALGENCAN/sources/algencan/inc \
> mv bmfeasgencan.mod $ALGENCAN/sources/algencan/inc \
```

Fonte: Autor

3.3.3 Configuração automatizada

Por meio da ferramenta GNU make é possível automatizar o processo de transformar código fonte em código executável (MECKLENBURG, 2004). A ferramenta make sabe quais passos seguir e quais não precisam ser realizados novamente para refazer o programa desejado cada vez que ela é executada (MECKLENBURG, 2004).

Um arquivo **Makefile** deve ser criado no diretório que contém os arquivos necessários para a geração do código executável. Nesse arquivo devem ser descritos os relacionamentos e *timestamps* dos arquivos utilizados para a geração do executável (MECKLENBURG, 2004).

No diretório raiz do ALGENCAN, **$ALGENCAN**, tem-se os seguintes comandos possíveis para a ferramenta make:

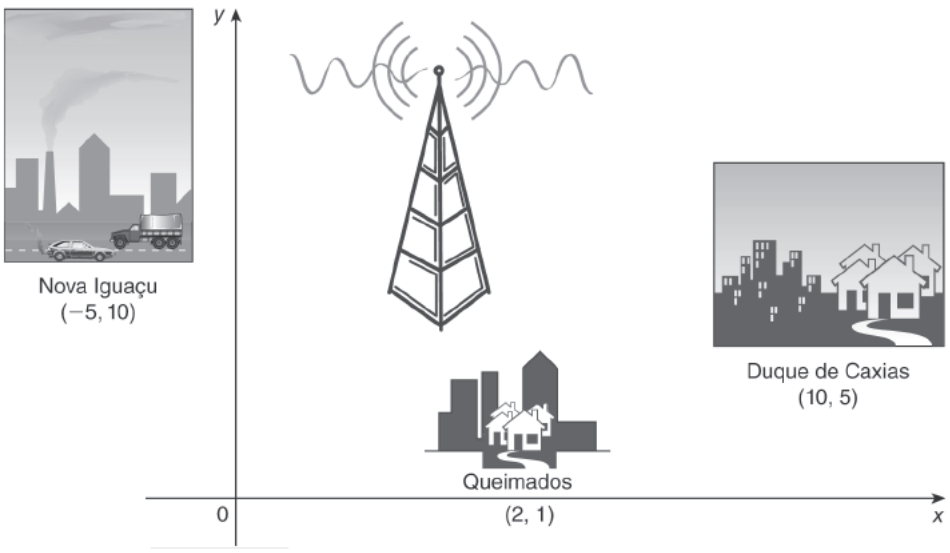
- make: realiza todos os passos anteriores de compilação das bibliotecas BLAS, HSL e ALGENCAN.
- make clean: remove todos os arquivos compilados e incluídos, exceto aqueles em diretórios **/lib**
- make distclean: faz o mesmo que **make clean**, mas remove também os arquivos em **/lib** e **.mod**.
- make example: compila o programa de exemplo em **$ALGENCAN/sources/examples/algencanma.f90** e gera o executável em **$ALGENCAN/bin/examples/algencanma**.

3.4 Exemplo de aplicação

Para demonstar o uso do ALGENCAN após a sua instalação e configuração, será abordado um problema de aplicação do dia a dia adaptado de Lachtermacher (2016).

O problema escolhido consiste em encontrar a melhor localização para a construção de uma antena de transmissão que fornecerá sinal de telefonia para três cidades. A antena possui a limitação técnica da impossibilidade de fornecer sinal há mais de 10 quilômetros do centro de cada cidade. A Figura 14 ilustra melhor o problema.

Figura 14 – Mapa das cidades e antena



Fonte: (LACHTERMACHER, 2016)

As localizações das cidades no plano cartesiano se encontram na Tabela 6.

Tabela 6 – Localização do centro das cidades

Localidade	X	Y
Nova Iguaçu	-5	10
Queimados	2	1
Duque de Caxias	10	5

Fonte: Autor

O primeiro passo é o entendimento e a modelagem do problema. Considerando que as localizações são definidas tendo como base o plano cartesiano, a localização da antena será representada por (X_1, X_2).

A equação que calcula a distância entre dois pontos $P1(x1, y1)$ e $P2(x2, y2)$ é dada por:

$$D_{P1 \to P2} = \sqrt{(\, x_1 - x_2)^{\,2} + (\, y_1 - y_2)^{\,2}}.$$

Dessa maneira, a distância entre a antena e a cidade de Duque de Caxias, por exemplo, é calculada como:

$$D_{P1 \to P2} = \sqrt{(\, 10 - X_1)^{\,2} + (\, 5 - X_2)^{\,2}}.$$

A função objetivo para esse problema é caracterizada como sendo a minimização da soma das distâncias entre a antena e cada uma das cidades. A função objetivo é representada como sendo:

$$\begin{aligned} f(X_1, X_2) = \; & \sqrt{(\, -5 - X_1)^{\,2} + (\, 10 - X_2)^{\,2}} \\ & + \sqrt{(\, 2 - X_1)^{\,2} + (\, 1 - X_2)^{\,2}} \\ & + \sqrt{(\, 10 - X_1)^{\,2} + (\, 5 - X_2)^{\,2}}. \end{aligned} \tag{3.4}$$

As restrições do problema são definidas como a limitação que a antena não pode estar localizada a mais de 10 quilômetros do centro de cada cidade e são representadas nas equações a seguir.

$$\begin{aligned} g_1(X_1, X_2) &= \sqrt{(\, -5 - X_1)^{\,2} + (\, 10 - X_2)^{\,2}} \leq 10, \\ g_2(X_1, X_2) &= \sqrt{(\, 2 - X_1)^{\,2} + (\, 1 - X_2)^{\,2}} \leq 10, \\ g_3(X_1, X_2) &= \sqrt{(\, 10 - X_1)^{\,2} + (\, 5 - X_2)^{\,2}} \leq 10. \end{aligned}$$

Reescrevendo as restrições da maneira aceita pelo ALGENCAN

$$g_1(X_1, X_2) = \sqrt{(-5 - X_1)^2 + (10 - X_2)^2} - 10 \leq 0, \tag{3.5}$$

$$g_2(X_1, X_2) = \sqrt{(2 - X_1)^2 + (1 - X_2)^2} - 10 \leq 0, \tag{3.6}$$

$$g_3(X_1, X_2) = \sqrt{(10 - X_1)^2 + (5 - X_2)^2} - 10 \leq 0. \tag{3.7}$$

As 5 sub-rotinas obrigatórias descritas na Seção 3.2 são escritas para esse problema como:

(a) `evalf`: A função objetivo possui somente duas variáveis, portanto, $n = 2$, e é representada por

$$\begin{aligned} f(X_1, X_2) = & \sqrt{(-5 - X_1)^2 + (10 - X_2)^2} \\ & + \sqrt{(2 - X_1)^2 + (1 - X_2)^2} \\ & + \sqrt{(10 - X_1)^2 + (5 - X_2)^2}. \end{aligned}$$

(b) `evalg`: o gradiente da função objetivo é uma matriz de ordem 2×1 da forma

$$\nabla f(X_1, X_2) = \begin{bmatrix} \left(\frac{X_1+5}{\sqrt{X_1^2+10X_1+X_2^2+125-20X_2}} + \frac{X_1-2}{\sqrt{X_1^2-4X_1+X_2^2+5-2X_2}} + \frac{X_1-10}{\sqrt{X_1^2-20X_1+X_2^2+125-10X_2}} \right) \\ \left(\frac{X_2-10}{\sqrt{X_2^2-20X_2+X_1^2+10X_1+125}} + \frac{X_2-1}{\sqrt{X_2^2-2X_2+X_1^2+5-4X_1}} + \frac{X_2-5}{\sqrt{X_2^2-10X_2+X_1^2+125-20X_1}} \right) \end{bmatrix}.$$

(c) `evalc`: a matriz c que armazena as restrições é de ordem 3×1 da forma

$$c = \begin{bmatrix} \left(\sqrt{(-5 - X_1)^2 + (10 - X_2)^2} - 10 \right) \\ \left(\sqrt{(2 - X_1)^2 + (1 - X_2)^2} - 10 \right) \\ \left(\sqrt{(10 - X_1)^2 + (5 - X_2)^2} - 10 \right) \end{bmatrix}.$$

(d) `evalj`: como existem apenas 3 restrições de desigualdade e 2 variáveis para esse problema, a matriz jacobiana do gradiente transposto das restrições é de ordem 3×2 e tem a forma

$$J = \begin{bmatrix} \left(\frac{X_1+5}{\sqrt{X_1+10X_1+X_2+125-20X_2}} & \frac{X_2-10}{\sqrt{X_1+10X_1+X_2+125-20X_2}} \right) \\ \left(\frac{X_1-2}{\sqrt{X_1-4X_1+X_2+5-2X_2}} & \frac{X_2-1}{\sqrt{X_1-4X_1+X_2+5-2X_2}} \right) \\ \left(\frac{X_1-10}{\sqrt{X_1-20X_1+X_2+125-10X_2}} & \frac{X_2-5}{\sqrt{X_1-20X_1+X_2+125-10X_2}} \right) \end{bmatrix}.$$

(e) `evalhl`: a Matriz Hessiana do Lagrangiano é dada por

$$
\begin{aligned}
&\nabla^2 L(x,\lambda,\mu) \\
&= \begin{bmatrix}
\begin{pmatrix}
\frac{X_2^2-20X_2+100}{(x_1^2+10x_1+X_2^2+125-20X_2)\sqrt{x_1^2+10x_1+X_2^2+125-20X_2}} \\
+\frac{X_2^2-2X_2+1}{(x_1^2-4x_1+X_2^2+5-2X_2)\sqrt{x_1^2-4x_1+X_2^2+5-2X_2}} \\
+\frac{X_2^2-10X_2+25}{(x_1^2-20x_1+125-10X_2)\sqrt{x_1^2-20x_1+125-10X_2}}
\end{pmatrix}
\begin{pmatrix}
-\frac{(x_1+5)(X_2-10)}{(x_1^2+10x_1+X_2^2+125-20X_2)^{\frac{3}{2}}} \\
-\frac{(x_1-2)(X_2-1)}{(x_1^2-4x_1+X_2^2+5-2X_2)^{\frac{3}{2}}} \\
-\frac{(x_1-10)(X_2-5)}{(x_1^2-20x_1+X_2^2+125-10X_2)^{\frac{3}{2}}}
\end{pmatrix} \\
\begin{pmatrix}
-\frac{(x_1+5)(X_2-10)}{(x_1^2+10x_1+X_2^2+125-20X_2)^{\frac{3}{2}}} \\
-\frac{(X_2-1)(x_1-2)}{(X_2^2-2X_2+x_1^2+5-4x_1)^{\frac{3}{2}}} \\
-\frac{(X_2-5)(x_1-10)}{(X_2^2-10X_2+x_1^2+125-20x_1)^{\frac{3}{2}}}
\end{pmatrix}
\begin{pmatrix}
\frac{x_1^2+10x_1+25}{(X_2^2-20X_2+x_1^2+10x_1+125)\sqrt{X_2^2-20X_2+x_1^2+10x_1+125}} \\
\frac{x_1^2-4x_1+4}{(X_2^2-2X_2+x_1^2+5-4x_1)\sqrt{X_2^2-2X_2+x_1^2+5-4x_1}} \\
\frac{x_1^2-20x_1+100}{(X_2^2-10X_2+x_1^2+125-20x_1)\sqrt{X_2^2-10X_2+x_1^2+125-20x_1}}
\end{pmatrix}
\end{bmatrix} \\
&+ \begin{bmatrix}
\begin{pmatrix}
\frac{X_2^2-20X_2+100}{(x_1^2+10x_1+X_2^2+125-20X_2)\sqrt{x_1^2+10x_1+X_2^2+125-20X_2}}
\end{pmatrix}
\begin{pmatrix}
-\frac{(x_1+5)(X_2-10)}{(x_1^2+10x_1+X_2^2+125-20X_2)^{\frac{3}{2}}}
\end{pmatrix} \\
\begin{pmatrix}
-\frac{(x_1+5)(X_2-10)}{(x_1^2+10x_1+X_2^2+125-20X_2)^{\frac{3}{2}}}
\end{pmatrix}
\begin{pmatrix}
\frac{x_1^2+10x_1+25}{(x_1^2+10x_1+X_2^2+125-20X_2)\sqrt{x_1^2+10x_1+X_2^2+125-20X_2}}
\end{pmatrix}
\end{bmatrix} \\
&+ \begin{bmatrix}
\begin{pmatrix}
\frac{X_2^2-2X_2+1}{(x_1^2-4x_1+X_2^2+5-2X_2)\sqrt{x_1^2-4x_1+X_2^2+5-2X_2}}
\end{pmatrix}
\begin{pmatrix}
-\frac{(x_1-2)(X_2-1)}{(x_1^2-4x_1+X_2^2+5-2X_2)^{\frac{3}{2}}}
\end{pmatrix} \\
\begin{pmatrix}
-\frac{(x_1-2)(X_2-1)}{(x_1^2-4x_1+X_2^2+5-2X_2)^{\frac{3}{2}}}
\end{pmatrix}
\begin{pmatrix}
\frac{x_1^2-4x_1+4}{(x_1^2-4x_1+X_2^2+5-2X_2)\sqrt{x_1^2-4x_1+X_2^2+5-2X_2}}
\end{pmatrix}
\end{bmatrix} \\
&+ \begin{bmatrix}
\begin{pmatrix}
\frac{X_2^2-10X_2+25}{(x_1^2-20x_1+X_2^2+125-10X_2)\sqrt{x_1^2-20x_1+X_2^2+125-10X_2}}
\end{pmatrix}
\begin{pmatrix}
-\frac{(x_1-10)(X_2-5)}{(x_1^2-20x_1+X_2^2+125-10X_2)^{\frac{3}{2}}}
\end{pmatrix} \\
\begin{pmatrix}
-\frac{(x_1-10)(X_2-5)}{(x_1^2-20x_1+X_2^2+125-10X_2)^{\frac{3}{2}}}
\end{pmatrix}
\begin{pmatrix}
\frac{x_1^2-20x_1+100}{(x_1^2-20x_1+X_2^2+125-10X_2)\sqrt{x_1^2-20x_1+X_2^2+125-10X_2}}
\end{pmatrix}
\end{bmatrix}
\end{aligned}
$$

.

Tendo as sub-rotinas calculadas basta editar o arquivo de exemplo do ALGENCAN, **$ALGENCAN/sources/examples/algencanma.f90**, com as equações para as sub-rotinas e fornecendo os parâmetros explicados na Seção 3.2. Para a inicialização do problema foram escolhidos os valores 0 e 0 para X_1 e X_2 respectivamente.

Compilando o programa exemplo com o comando **make example** será gerado um arquivo binário na pasta **$ALGENCAN/bin/examples**. Após compilado basta executar o binário com o comando **./bin/example/algencanma**. O resultado pode ser visto na Figura 15.

Figura 15 – Solução do problema pelo ALGENCAN

```
Number of variables                                =            2
Number of equality constraints                     =            0
Number of inequality constraints                   =            3

(REPORTED BY SOLVER) istop                         =            2
(REPORTED BY SOLVER) ierr                          =            0
(REPORTED BY SOLVER) f                             =    20.101228774698491
(REPORTED BY SOLVER) csupn                         =    2.7220892206969438E-011
(REPORTED BY SOLVER) ssupn                         =    0.0000000000000000
(REPORTED BY SOLVER) nlpsupn                       =    4.3728162868461595E-008
(REPORTED BY SOLVER) bounds violation              =    0.0000000000000000
(REPORTED BY SOLVER) Number of outer iterations    =            7
(REPORTED BY SOLVER) Number of inner iterations    =           56
(REPORTED BY SOLVER) Number of Newton-KKT trials   =            7
(REPORTED BY SOLVER) Number of Newton-KKT iterations  =         65

(COMPUTED BY CALLER) Number of calls to evalf      =          168
(COMPUTED BY CALLER) Number of calls to evalg      =          144
(COMPUTED BY CALLER) Number of calls to evalc      =          256
(COMPUTED BY CALLER) Number of calls to evalj      =          139
(COMPUTED BY CALLER) Number of calls to evalhl     =          121
(COMPUTED BY CALLER) CPU time in seconds           =    1.0984999999999998E-002
x values =    2.3392270737943202        3.2076700639744495
f(x) =    20.101228774698491

(COMPUTED BY CALLER) f                             =    20.101228774698491
(COMPUTED BY CALLER) csupn                         =    2.7220892206969438E-011
(COMPUTED BY CALLER) bounds violation              =    0.0000000000000000

When a quantity appears as computed by solver and computed by caller, they must coincide.
(In case they do not coincide, please report it as a bug.)
Note: The following floating-point exceptions are signalling: IEEE_INVALID_FLAG IEEE_DENORMAL
maiconmares@maiconmares-Inspiron-3501:~/Desktop/algencan-4.0.0$
```

Fonte: Autor

Como pode ser visto, tem-se como solução para $f(X_1, X_2)$ a soma das distâncias de aproximadamente 20.1 unidades e os valores de X_1 e X_2 que nos levam a essa minimização são 2.34 e 3.2, respectivamente.

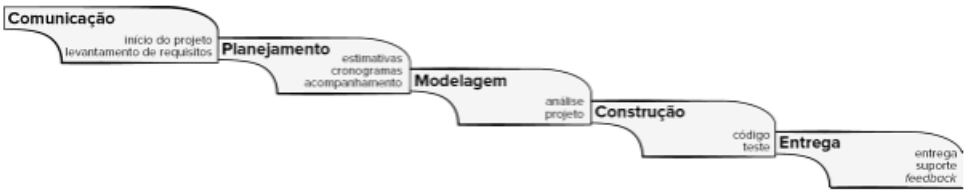
O código completo para o exemplo aqui abordado pode ser encontrado no repositório [5].

[5] https://github.com/MaiconMares/algencan-exercises

4 Metodologia

Para o desenvolvimento da interface foi escolhido o modelo de processo procedural cascata. O modelo cascata propõe uma abordagem sistemática e sequencial para o desenvolvimento de *software* (PRESSMAN; MAXIM, 2021). O processo inicia com a coleta de requisitos passando pelo planejamento, modelagem, construção (desenvolvimento e testes) e entrega, finalizando com o consequente suporte ao *software* desenvolvido (PRESSMAN; MAXIM, 2021).

Figura 16 – O modelo cascata



Fonte: (PRESSMAN; MAXIM, 2021)

Apesar das críticas relativas ao atraso e custos adicionais em projetos de *software* que adotaram o modelo cascata (PRASETYA; SUHARJITO; PRATAMA, 2021), esse ainda se mostra adequado e eficiente em contextos específicos.

No trabalho de Thesing, Feldmann e Burchardt (2021) é proposto um modelo de decisão dividido em duas etapas que auxilia na escolha entre processos de desenvolvimento que seguem o cascata clássico ou métodos ágeis.

Na primeira etapa a escolha do processo é realizada analisando critérios de exclusão, sendo (THESING; FELDMANN; BURCHARDT, 2021):

- Falta de decomposição: não é possível separar o resultado final em entregáveis separados
- Desenvolvimento em apenas uma rodada: uma abordagem iterativa com mudanças frequentes ou passo a passo não é possível do ponto de vista técnico ou legal ou está associada com custos impraticáveis.
- Criticidade do projeto: riscos operacionais não permitem uma abordagem ágil iterativa.

Na segunda etapa são pontuados 15 critérios divididos nas categorias de escopo, tempo, custos, contexto organizacional e características da equipe (THESING; FELDMANN; BURCHARDT, 2021). Se um ou mais critérios de exclusão se aplicarem ao

projeto, o seu sucesso com métodos ágeis é improvável (THESING; FELDMANN; BURCHARDT, 2021).

4.0.1 Seleção do processo para o desenvolvimento da interface

Aplicando o Modelo de Decisão (THESING; FELDMANN; BURCHARDT, 2021) na escolha do processo para o desenvolvimento da interface proposta no presente trabalho, o processo mais adequado é o processo cascata.

Foi feita essa escolha devido à implementação da interface se encaixar nos dois critérios de exclusão, falta de decomposição e desenvolvimento em apenas uma rodada, entre os 3 inicialmente apresentados.

Para o desenvolvimento da interface para o ALGENCAN é necessário que sejam implementadas todas as rotinas responsáveis pelos cálculos parciais. Após feitos os cálculos parciais, é possível chamar a rotina principal que então resolve o problema de otimização.

Desse modo, só é possível testar a correta implementação das rotinas e a solução do problema de otimização após termos a interface para cada uma das rotinas. Logo, não é possível decompor o desenvolvimento em partes menores entregáveis.

Ainda ocorre a dependência entre as rotinas que realizam os cálculos parciais. Somando esse fato com a dependência já citada, não é possível utilizar uma abordagem iterativa que realize mudanças frequentes sem que se tenha que alterar grande parte do projeto. Portanto, o critério de desenvolvimento em apenas uma rodada também se aplica.

4.0.2 Emprego do Processo Cascata no desenvolvimento da interface

O modelo cascata adotado foi modificado para o contexto deste trabalho e as fases bem como o que será realizado em cada uma delas são:

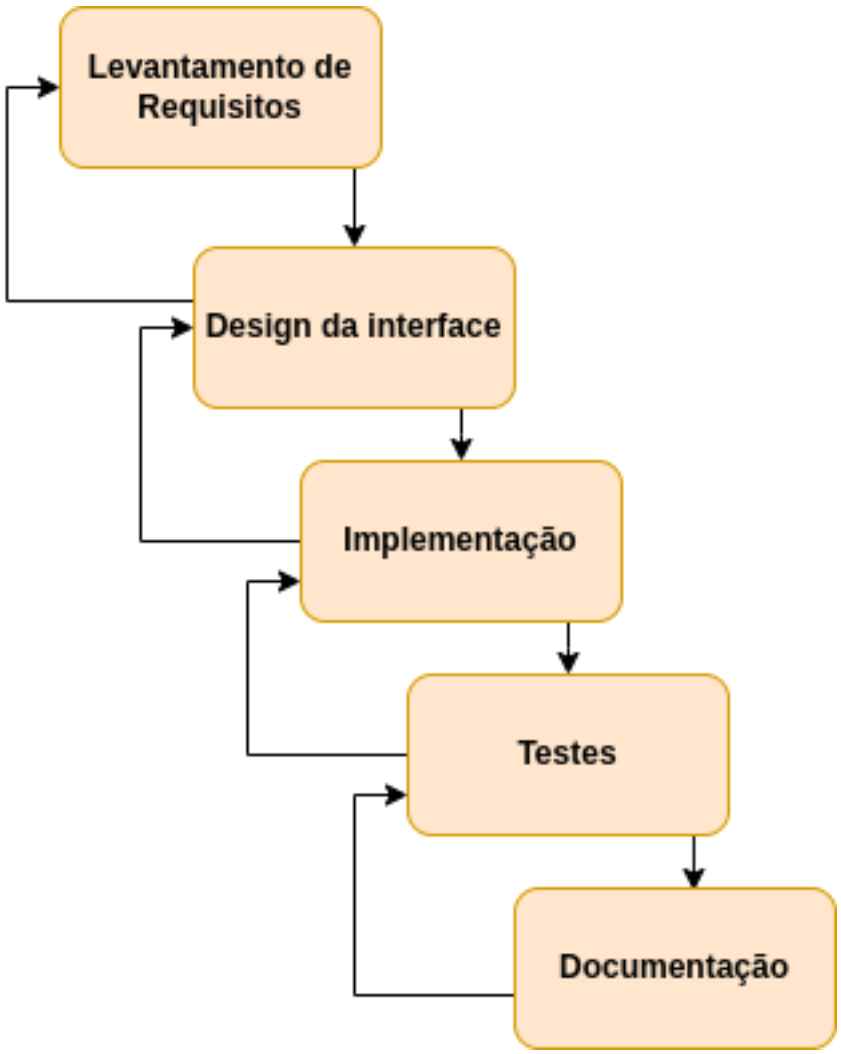
- Levantamento de requisitos: os requisitos necessários serão coletados a partir do uso do ALGENCAN (testes manuais) aplicado a um problema de otimização, da pesquisa bibliográfica sobre os recursos de interoperabilidade com Fortran presentes na Linguagem Julia e das reuniões de orientação.
- *Design* da interface: nesta etapa será projetada a estrutura da interface, isto é, as funções, os métodos e a interação entre esses.
- Implementação: a estrutura da interface projetada na fase anterior será de fato desenvolvida, ou seja, a construção do *software*.
- Testes: será escolhido um conjunto de problemas a serem resolvidos e será avaliada a sua correta resolução, bem como será comparado o tempo necessário para a sua

solução utilizando a interface com o tempo gasto utilizando somente a biblioteca em Fortran.

- Documentação: todo o código desenvolvido, a estrutura da interface e como utilizá-la serão documentados e disponibilizados em repositório aberto.

O modelo modificado ainda prevê retornos, caso sejam necessários, à fase anterior para correções pontuais. A estrutura completa pode ser melhor compreendida na figura abaixo.

Figura 17 – Cascata modificado



Fonte: Autor

4.1 Fase de Testes

Na Fase de Testes, a fim de validar a interoperabilidade desenvolvida, foi utilizado o pacote CUTEst. O pacote oferece uma *Application Programming Interface* (API) para

a linguagem Julia e abrange cerca de 1150 exemplos, entre eles, envolve problemas não lineares sem restrições, problemas de mínimos quadrados e, apesar de ser focado em problemas sem restrições, também oferece problemas de otimização não lineares com restrições (GOULD; ORBAN; TOINT, 2015).

O CUTEst fornece diversos métodos que permitem a manipulação do problema em Julia e que pode ser combinado com os métodos de resolução de derivadas já embutidos na linguagem, tornando mais produtivo o trabalho do usuário (GOULD; ORBAN; TOINT, 2015).

Para o cálculo das derivadas, das restrições e outros realizados pelas funções de avaliação, foi escolhido o pacote *Non-Linear Programming Models* (NLPModels) implementado em Julia. O pacote fornece métodos que recebem o problema no formato descrito pelo CUTEst e realizam todos os cálculos necessários sem que o usuário tenha que implementar uma única função (ORBAN; SIQUEIRA; contributors, 2020).

Para a validação da interoperabilidade serão adotados os passos a seguir:

1. Selecionar a partir do pacote CUTEst um conjunto de problemas de otimização não linear com restrições.
2. Desenvolver programa para conduzir os testes.
3. Mensurar o tempo, em uma máquina pré-definida, para resolver os problemas nos dois cenários: utilizando a interoperabilidade junto ao ALGENCAN e somente o ALGENCAN.
4. Comparar os tempos de execução em cada um dos cenários.

5 Implementação da Interface

É reconhecido entre a maioria dos especialistas e praticantes da indústria que atividades relacionadas aos requisitos de software, se mal executadas, levam à vulnerabilidade de projetos de software (BOURQUE; FAIRLEY; SOCIETY, 2014). Tendo em vista a importância da correta definição dos requisitos, foram identificadas as fontes dos requisitos e técnicas para o seu levantamento. Cada uma delas é descrita a seguir.

Fontes de requisitos:

- Reuniões de orientação
- Revisão bibliográfica
- Testes manuais sobre o ALGENCAN

Técnicas de levantamento:

- *Brainstorming*: foram realizadas discussões abertas durante as reuniões de orientação, sobre o que deveria ser implementado na interface e melhores abordagens, que auxiliaram na coleta dos requisitos.
- Análise de literatura: revisão de documentação, artigos e trabalhos similares. Por meio desses foi possível a extração de requisitos, tais como boas práticas, restrições e compatibilidade de tipos.
- Introspecção: baseado nas experiências prévias, na perspectiva do responsável pela implementação e nos objetivos do trabalho foram estimadas as propriedades que a interface deveria apresentar.

É importante ressaltar que apesar dessa atividade ser minuciosamente executada no início do processo, alguns requisitos são elicitados durante outras fases, estando essa atividade presente em outras fases. Os requisitos funcionais e não funcionais levantados são apresentados na Tabela 7 e na Tabela 8, respectivamente.

Tabela 7 – Requisitos Funcionais

ID	Descrição do Requisito	Prioridade	Dependência
RF01	A interface deve ser capaz de solucionar problemas de otimização não linear com e sem restrições	Alta	RNF03,RNF04
RF02	A interface deve ser capaz de fazer chamada e repassar as funções de avaliação e parâmetros do problema à biblioteca do ALGENCAN	Alta	RNF03,RNF04
RF03	Após solucionado um problema, deve ser apresentado o valor da função objetivo encontrado e o tempo gasto em segundos	Alta	RF02

Fonte: Autor

Tabela 8 – Requisitos Não Funcionais

ID	Descrição do Requisito	Categoria
RNF01	Os tipos de dados utilizados na interface em Julia devem ser compatíveis com os dados a serem recebidos pelo ALGENCAN em Fortran	Compatibilidade
RNF02	A interface deve fornecer as funções de avaliação com assinaturas compatíveis às esperadas pelo ALGENCAN	Compatibilidade

Fonte: Autor

5.1 *Design* da Interface

Nessa etapa, foi pensado a relação entre os módulos e o fluxo de execução da interface. O fluxo de execução é constituído por 3 etapas:

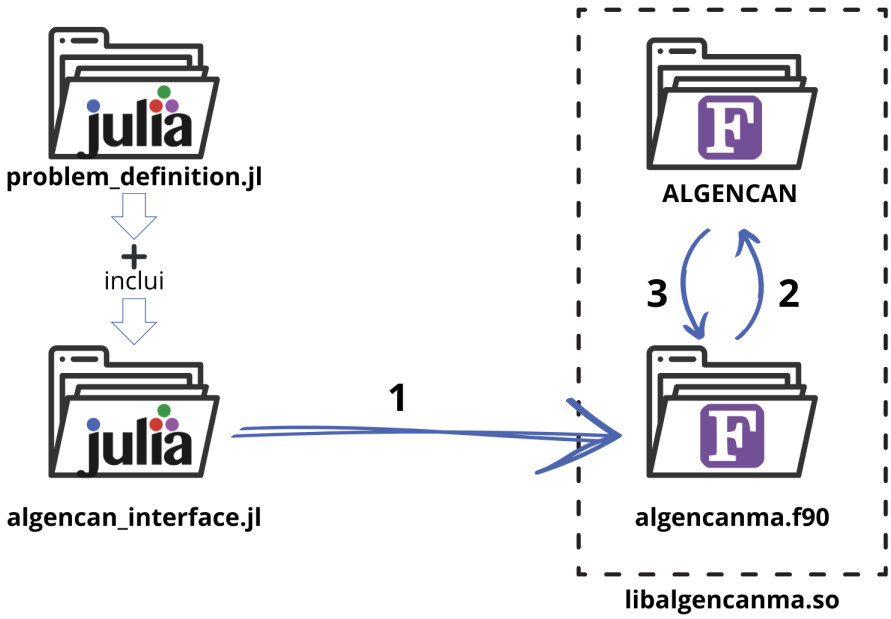
1. O programa `algencan_interface.jl` inclui o módulo, `problem_definition.jl`, onde as rotinas de avaliação e parâmetros que descrevem o problema estão definidos. O fluxo inicia com a chamada ao programa `algencanma.f90`. Na chamada são passadas as variáveis e funções de avaliação esperadas pelo ALGENCAN.

2. O programa `algencanma.f90` recebe todas as variáveis e funções para o problema em questão, inicializa variáveis locais e faz chamada e repassa todos esses dados à rotina do ALGENCAN.

3. A rotina do ALGENCAN irá receber as variáveis e funções, solucionar o problema e retornar as variáveis com os valores atualizados para o `algencanma.f90` que, por fim, apresenta o resultado ao usuário e encerra a execução.

O fluxo de execução com as etapas e os arquivos envolvidos nessas pode ser melhor visualizado na Figura 18. É importante destacar que o arquivo `algencanma.f90` é compilado junto às rotinas e dependências do ALGENCAN para gerar a biblioteca compartilhada `libalgencanma.so`.

Figura 18 – Fluxo de execução da interface em Julia



Fonte: Autor

5.2 Implementação

A interface é composta por 2 arquivos:

- `problem_definition.jl`: responsável pela definição do problema. É nesse arquivo onde devem ser implementadas as 5 funções de avaliação obrigatórias e os parâmetros com seus valores iniciais para o problema. Tais funções e parâmetros são resumidas no Código 5.1. As funções de avaliação podem ser nomeadas da maneira que o usuário desejar.
- `algencan_interface.jl`: o código completo está definido no Código 5.2. Recebe as funções e parâmetros definidos e exportados no arquivo `problem_definition.jl` nas linhas 1 e 3. Carrega a biblioteca compartilhada na linha 8. Então, define os tipos e assinatura das funções, nas linhas 12 a 31, e faz chamada, e os repassa à rotina `init` implementada no arquivo `algencanma.f90` nas linhas 33 a 44.

```
1 module ProblemDefinition
2   export MyDataPtr,evalf!,evalc!,evalg!,evalj!,evalhl!,
    problem_params
3
4   mutable struct MyDataPtr
5     counters::NTuple{5, Int32}
```

```
6
7       MyDataPtr(counters) = new(counters)
8     end
9
10    function problem_params()::Tuple
11      ...
12
13      return x,n,f,g,c,lind,lbnd,uind,ubnd,m,p,lambda,jnnzmax,
      hlnnzmax,epsfeas,epscompl,epsopt,
14            rhoauto,rhoini,scale,extallowed,corrin,inform,ind,
      pdata
15    end
16
17    function evalf!(n::Int32,x,f::Ptr{Float64},inform::Int32,
      pdataptr::Ptr{MyDataPtr}=nothing)::Nothing
18        ...
19    end
20
21    function evalg!(n::Int32,x,g,inform::Int32,pdataptr::MyDataPtr=
      nothing)::Nothing
22        ...
23    end
24
25    function evalc!(
26      n::Int32,x,m::Int32,p::Int32,c,inform::Int32,pdataptr::
      MyDataPtr=nothing
27      )::Nothing
28        ...
29    end
30
31    function evalj!(n::Int32,x::Ptr{Float64},m::Int32,p::Int32,ind
      ::Ptr{Int32},
32      sorted::Ptr{Int32},jsta::Ptr{Int32},jlen::Ptr{Int32},lim::
      Int32,
33      jvar::Ptr{Int32},jval::Ptr{Float64},inform::Int32,pdataptr::
      MyDataPtr=nothing
34      )::Nothing
35        ...
36    end
37
38    function evalhl!(
39      n::Int32,x::Ptr{Float64},m::Int32,p::Int32,lambda::Ptr{
```

```
    Float64},lim::Int32,
40    inclf::Int32,hlnnz::Ptr{Int32},hlrow::Ptr{Int32},hlcol::Ptr{
    Int32},hlval::Ptr{Float64},
41    inform::Int32,pdataptr::MyDataPtr=nothing
42    )::Nothing
43
44    ...
45  end
46 end
```

Código 5.1 – Definição do problema

```
1  include("./problem_definition.jl")
2
3  using .ProblemDefinition
4
5  using Libdl
6
7  curr_dir = Vector{String}(["./"])
8  lib_path = Libdl.find_library("libalgencanma.so", curr_dir)
9
10 function run_algencan()::Vector{Float64}
11   lib = Libdl.dlopen(lib_path, RTLD_NOW|RTLD_GLOBAL)
12   evalf_ptr = @cfunction(evalf!, Nothing, (Ref{Int32},Ptr{Float64
     },Ptr{Float64},Ref{Int32},Ptr{MyDataPtr}))
13   evalg_ptr = @cfunction(evalg!, Nothing, (Ref{Int32},Ptr{Float64
     },Ptr{Float64},Ref{Int32},Ptr{MyDataPtr}))
14   evalc_ptr = @cfunction(evalc!, Nothing, (
15     Ref{Int32},Ptr{Float64},Ref{Int32},Ref{Int32},Ptr{Float64},
     Ref{Int32}, Ptr{MyDataPtr}
16   ))
17
18   evalj_ptr = @cfunction(evalj!, Nothing, (
19     Ref{Int32},Ptr{Float64},Ref{Int32},Ref{Int32},Ptr{Int32},
20     Ptr{Int32},Ptr{Int32},Ptr{Int32},Ref{Int32},
21     Ptr{Int32},Ptr{Float64},Ref{Int32},Ptr{MyDataPtr}
22   ))
23
24   evalhl_ptr = @cfunction(evalhl!, Nothing, (
25     Ref{Int32},Ptr{Float64},Ref{Int32},Ref{Int32},Ptr{Float64},
     Ref{Int32},
26     Ref{Int32},Ptr{Int32},Ptr{Int32},Ptr{Int32},Ptr{Float64},
27     Ptr{Int32},Ptr{MyDataPtr}
```

```
28    ))
29
30    x,n,f,lind,lbnd,uind,ubnd,m,p,lambda,jnnzmax,hlnnzmax,epsfeas,
      epscompl,epsopt,
31            rhoauto,rhoini,scale,extallowed,corrin,inform,maxoutit,
      pdata = problem_params()
32
33    @ccall lib_path.__algencanma_MOD_init(
34      evalf_ptr::Ptr{Cvoid}, evalg_ptr::Ptr{Cvoid},
35      evalc_ptr::Ptr{Cvoid}, evalj_ptr::Ptr{Cvoid},
36      evalhl_ptr::Ptr{Cvoid}, x::Ptr{Float64},
37      n::Ref{Int32},f::Ref{Float64},lind::Ptr{Int32}, lbnd::Ptr{
      Float64},
38      uind::Ptr{Int32}, ubnd::Ptr{Float64},
39      m::Ref{Int32}, p::Ref{Int32}, lambda::Ptr{Float64},
40      jnnzmax::Ref{Int32}, hlnnzmax::Ref{Int32}, epsfeas::Ref{
      Float64},
41      epscompl::Ref{Float64},epsopt::Ref{Float64}, rhoauto::Ref{
      Int32},
42      rhoini::Ref{Float64},scale::Ref{Int32},extallowed::Ref{Int32
      },corrin::Ref{Int32},
43      inform::Ref{Int32},maxoutit::Ref{Int32},pdata::Ref{MyDataPtr}
44      )::Cvoid
45
46    Libdl.dlclose(lib)
47
48    return x
49  end
```

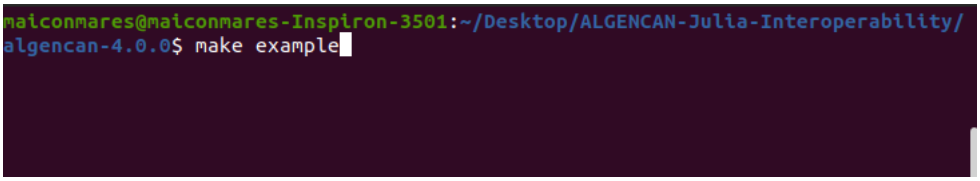
Código 5.2 – Carregamento da biblioteca compartilhada e chamada ao ALGENCAN

5.2.1 Utilização da Interface

Para executar a interface é necessário a instalação da linguagem Julia a partir da versão 1.5. Após instalada, é preciso compilar a biblioteca compartilhada com auxílio do `make` como apresentado na Figura 19. Tendo finalizado o processo de compilação, será gerado o arquivo `libalgencanma.so` para a biblioteca compartilhada.

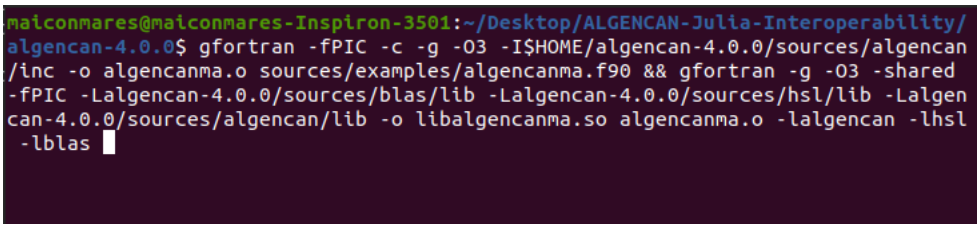
Outra opção é compilar manualmente conforme a Figura 20.

Figura 19 – Compilação da biblioteca compartilhada



Fonte: Autor

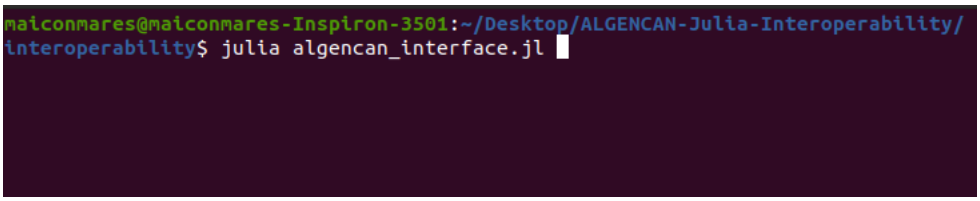
Figura 20 – Compilação da biblioteca compartilhada



Fonte: Autor

Logo após a compilação, apenas é necessário que se execute o arquivo `algencan_interface.jl` como demonstra a Figura 21.

Figura 21 – Execução da interface em Julia



Fonte: Autor

5.3 Testes

A realização dos testes consiste em definir um conjunto de problemas e para esses coletar os dados de tempo de execução e valor da função objetivo. Esses dados são coletados para os problemas resolvidos pela interface em Julia e diretamente pela biblioteca do ALGENCAN.

O conjunto de problemas consiste em 17 problemas com as seguintes características:

- Mínimo de 20 variáveis e máximo de 50 variáveis.
- Função objetivo deve ser não linear.

- Apenas problemas com restrições de desigualdade da forma $c(x) \leq u$ ou $c(x) \geq \ell$. Problemas com restrições da forma $\ell \leq c(x) \leq u$ não serão considerados.

Os valores mínimo e máximo de variáveis foram escolhidos visando testar o máximo de caminhos que podem ser percorridos em código pela interface. O pacote CUTEst fornece 6 tipos de função objetivo. Foram escolhidos os tipos quadrática e soma de quadrados devido ao interesse do trabalho serem os problemas não lineares.

Ao testar problemas, oriundos do CUTEst, em que a mesma restrição está presente nos limites inferior e superior observou-se que a restrição aparece somente uma vez no vetor de restrições. Antes de repassar as restrições no formato esperado pelo ALGENCAN, é necessária a duplicação de tal restrição e que a restrição para o limite inferior tenha seu sinal invertido, pois o ALGENCAN suporta somente o operador $\leq$ como já mencionado anteriormente. Também se fazem necessárias modificações na jacobiana das restrições e na matriz hessiana. Entretanto, o pacote NLPModels não permite tais modificações de modo automático e a escrita manual das funções de avaliação para o cálculo da jacobiana e da hessiana são necessários. Dar suporte a esse tipo de restrições está fora do escopo deste trabalho e ficará como recomendação de trabalho futuro.

Para guiar os testes na interface em Julia foram desenvolvidos os arquivos:

- `Julia_interface_4_CUTEst.jl`: seu propósito é servir como interface para o pacote CUTEst. Define as funções de avaliação e os parâmetros iniciais do problema, de modo genérico, para receber qualquer problema do CUTEst. Disponível no Apêndice A.

- `Julia_interface_4_CUTEst_test.jl`: faz inclusão dos módulos definidos nos arquivos `Julia_interface_4_CUTEst.jl` e `algencan_interface.jl`. Define o conjunto de testes, percorre cada um dos problemas e define globalmente o problema atual. Ainda faz chamada e repassa os parâmetros iniciais e funções de avaliação do problema em questão para `algencan_interface.jl`. Então, mensura o tempo gasto para resolver o problema, coleta o valor da função objetivo encontrado e salva esses dados em um arquivo de resultados do tipo `.txt`. Disponível no Apêndice B.

Para persistir os valores em parâmetros e para acessar endereços de memória de vetores passados pelo ALGENCAN para as funções definidas em Julia foram utilizados 2 métodos da linguagem Julia:

- `unsafe_wrap(Array, pointer::PtrT, dims; own = false)`: envolve um objeto em Julia do tipo *array* em torno do endereço fornecido pelo ponteiro `pointer` sem realizar uma cópia. O tipo dos elementos de `pointer` são definidos por `T` e sua

dimensão por `dims`. O último parâmetro, `own`, determina se Julia deve tomar controle da memória enquanto realiza o acesso.

- `unsafe_store!(p::PtrT, x, i::Integer=1)`: armazena o valor de `x` no endereço do índice `i` em `p`.

Para iniciar os testes sobre a interface em Julia basta executar o comando conforme a Figura 22. Após finalizados os testes é gerado um arquivo nomeado `julia_interface_4_CUTEst_results.txt`. Neste arquivo, em cada linha, é apresentado o nome do problema, o tempo gasto para sua resolução e o valor da função objetivo encontrado.

Figura 22 – Execução dos testes em Julia

```
maiconmares@maiconmares-Inspiron-3501:~/Desktop/ALGENCAN-Julia-Interoperability/interop
erability$ sudo julia tests/julia_interface_4_CUTEst_test.jl
[ Info: using problem repository
   ENV["MASTSIF"] = "/root/.julia/artifacts/a7ea0d0aaf29a39ca0fe75588fc077cdd5b5ed54/o
ptrove-sif-99c5b38e7d03"
 Starting allocation...
 outiter =            0  csupn =    0.0000000000000000       bdsvio =    0.000000000000
0000       ssupn =    0.0000000000000000       nlpsupn =    2118.0673505373147
 f =    704.10733409165073
 kkt:     F   1.2329254043486805E-316   6.9035569201118708E-310   1.0895815456419846E-3
16   0.0000000000000000        0.0000000000000000
 feas:    T   704.10733409165073        0.0000000000000000        0.0000000000000000
     0.0000000000000000        2118.0673505373147
 infeas:  F   0.0000000000000000        1.4908328097605279E-316   1.4908355765281447E-3
16   0.0000000000000000       -4.4258657827755827E-163
 best:      704.10733409165073        0.0000000000000000        0.0000000000000000
   0.0000000000000000        2118.0673505373147
 Trying Newton-KKT.
 newtkktb: iter =            0  csupn =    0.0000000000000000       bdsvio =    0.00000
00000000       ssupn =    0.0000000000000000       nlpsupn =    2118.0673505373147
```

Fonte: Autor

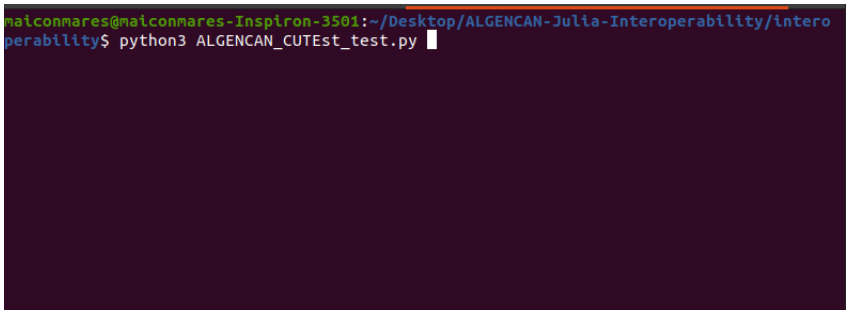
Para a execução dos testes sobre o ALGENCAN diretamente em Fortran foram utilizados 2 arquivos:

- `algencanma-forcutest.f90`: interface para o CUTEst em Fortran. Essa interface já se encontrava implementada e está disponível no site oficial do projeto TANGO[1].
- `ALGENCAN_CUTEst_test.py`: lê o conjunto de problemas escolhido e para cada problema compila os arquivos necessários junto ao arquivo da interface e gera um arquivo binário. Então, executa o binário que irá solucionar o problema em questão. Após a resolução do problema, são coletados os dados de tempo e valor da função objetivo. Esses dados, junto a outros gerados pelo ALGENCAN, são salvos em um arquivo nomeado `ALGENCAN_CUTEst_results.txt`

Para iniciar os testes sobre o ALGENCAN basta executar o comando conforme a Figura 23.

[1] https://www.ime.usp.br/ egbirgin/tango/downloads.php#form

Figura 23 – Execução dos testes em Fortran



Fonte: Autor

5.4 Documentação

A documentação para a interface em Julia, incluindo a interface em Julia com o CUTEst para os testes, foi gerada utilizando o pacote `Documenter.jl`. A documentação contém a descrição dos módulos desenvolvidos, o propósito de cada função e uma explicação sobre sua assinatura. Foi utilizado o GitHub Pages para hospedar a documentação[2].

[2] https://maiconmares.github.io/ALGENCAN-Julia-Interoperability/

6 Análise dos resultados

Durante o desenvolvimento incertezas ocorreram devido a não ser apontado precisamente onde os erros ocorreram e nem o tipo do erro. Os principais desafios encarados estavam relacionados ao processo de compilação e ao *layout* de memória.

Uma das dificuldades iniciais foi quanto à geração da biblioteca compartilhada. Ao compilar o ALGENCAN e as suas dependências, alguns módulos que dependem de rotinas provenientes de outros módulos não conseguiam encontrá-las. Toda rotina que pertence a uma biblioteca compartilhada é representada por um símbolo único gerado pelo compilador. Alguns símbolos não eram exportados devido a ordem incorreta das flags de compilação e da linkagem das dependências no processo de compilação.

Outro problema também relativo à compilação foi enfrentado. No arquivo `algencanma.f90`, quando havia alguma diferença na assinatura de funções passadas ao ALGENCAN, após compilar a biblioteca compartilhada, essa não é mais enxergada pela interface em Julia. Apenas o erro que o símbolo referente à rotina alvo da chamada não existe é informado, mas sua causa é incerta.

Quando algum parâmetro é definido com seu tipo diferente do esperado pelo ALGENCAN diversas inconsistências são geradas na interface. Foi observado que muitas vezes outros parâmetros definidos corretamente são afetados, como se seu espaço de memória fosse comprometido. Para problemas relativamente simples, com poucas variáveis, a interface continuava a funcionar mesmo quando algum parâmetro estava definido incorretamente, entretanto, ao iniciar os testes com problemas maiores as inconsistências surgiam.

Após a resolução dos problemas encarados durante o desenvolvimento da interface, foram iniciados os testes. Como já mencionado, foi necessário o desenvolvimento de uma interface com o CUTEst em Julia para execução dos testes automatizados.

Os testes foram executados no computador Dell Inspiron 15 3501, com processador Intel® Core™ i7-1165G7Os e 8 *gigabytes* de *Random Access Memory* (RAM). O sistema operacional utilizado foi o Ubuntu 20.04.6 *Long Term Support* (LTS). Os dados coletados de teste para a interface em Julia e os dados coletados executando diretamente o ALGENCAN por meio da interface em Fortran com o CUTEst são apresentados na Tabela 9.

Tabela 9 – Comparação do tempo de execução para problemas rodados com ALGENCAN nativamente em Fortran e usando a interface em Julia. Na tabela, $f(x^*)$ representa o valor final da função objetivo e é considerado apenas para mostrar que ambas convergiram marginalmente à mesma solução.

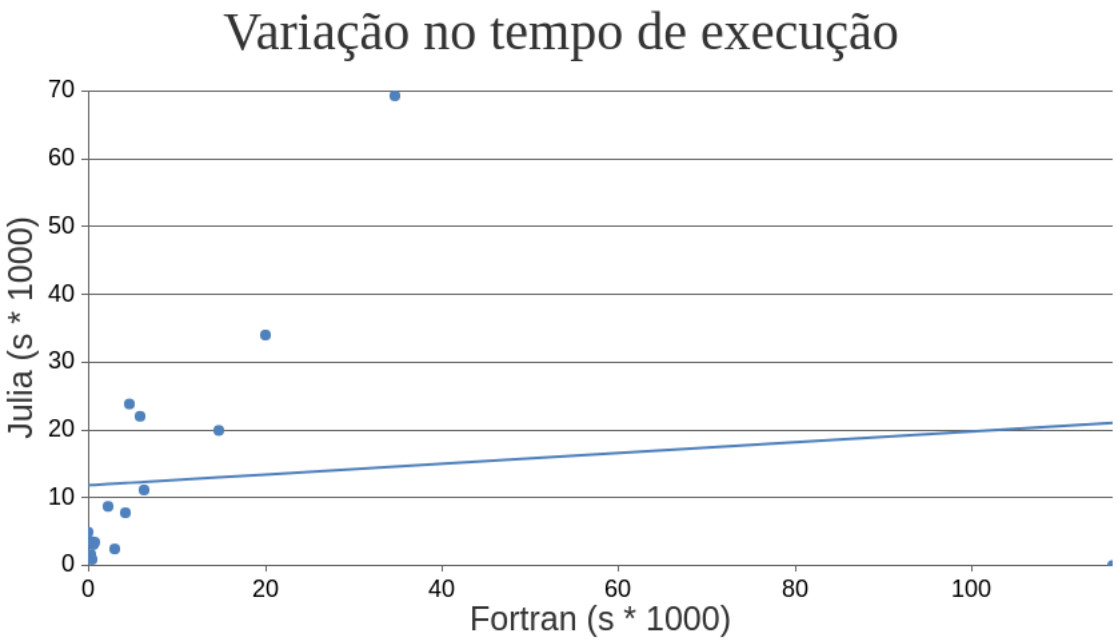
	ALGENCAN via Julia		**ALGENCAN via Fortran**	
Problema	**Tempo(s)**	$\boldsymbol{f(x^*)}$	**Tempo(s)**	$\boldsymbol{f(x^*)}$
ANTWERP	0.069322511	$3.245{\cdot}10^{+03}$	0.034710	$3.245{\cdot}10^{+03}$
HATFLDC	0.000901123	$3.418{\cdot}10^{-27}$	0.000460	$3.418{\cdot}10^{-27}$
ORTHREGB	0.002438293	$1.950{\cdot}10^{-30}$	0.002959	$1.950{\cdot}10^{-30}$
KSIP	646.309765009	0.575	0.115941	0.575
BQPGABIM	0.003509486	$-3.790{\cdot}10^{-05}$	0.000505	$-3.790{\cdot}10^{-05}$
OPTCNTRL	0.019930025	$5.499{\cdot}10^{+02}$	0.014731	$5.499{\cdot}10^{+02}$
SANTALS	0.007780265	$1.224{\cdot}10^{-05}$	0.004181	$1.224{\cdot}10^{-05}$
ERRINRSM	0.022021319	$3.772{\cdot}10^{+01}$	0.005797	$3.772{\cdot}10^{+01}$
BQPGASIM	0.003088735	$-5.519{\cdot}10^{-05}$	0.000596	$-5.519{\cdot}10^{-05}$
GOULDQP1	0.0111508	$-3.485{\cdot}10^{+03}$	0.006262	$-3.485{\cdot}10^{+03}$
ACOPP14	0.034014597	$8.081{\cdot}10^{+03}$	0.020113	$8.081{\cdot}10^{+03}$
ERRINROS	0.008715577	$4.040{\cdot}10^{+01}$	0.002298	$4.040{\cdot}10^{+01}$
HATFLDGLS	0.003427841	$6.922{\cdot}10^{-22}$	0.000791	$6.922{\cdot}10^{-22}$
CHNROSNB	0.023837534	$1.839{\cdot}10^{-26}$	0.004665	$1.973{\cdot}10^{-24}$
TOINTQOR	0.001456287	$1.175{\cdot}10^{+03}$	0.000244	$1.175{\cdot}10^{+03}$
3PK	0.00162304	1.720	0.000339	1.720
METHANB8LS	0.004924088	$1.564{\cdot}10^{-25}$	0.001158	$1.626{\cdot}10^{-25}$

Fonte: Autor

O gráfico de dispersão apresentado na Figura 24 permite uma melhor visualização da diferença nos tempos de execução entre a interface em Julia chamando o ALGENCAN (eixo Y) e o ALGENCAN executado nativamente em Fortran (eixo X). Todos os valores foram multiplicados por 1000 a fim de melhorar a visualização. O tempo de execução para o problema KSIP foi removido, pois este é um *outlier* e estava dificultando a visualização dos dados. Conforme os valores de execução para os problemas resolvidos pelo ALGENCAN via Fortran aumentam, os valores encontrados para a interface em Julia tendem a aumentar.

Em todos os problemas testados houve um aumento significativo do tempo necessário para a sua resolução quando realizada por meio da interface em Julia. O principal fator que contribuiu para esse aumento se deve às múltiplas chamadas que o ALGENCAN realiza às funções definidas em Julia e passadas como *callback*. Ao receber cada função como *callback*, na verdade, está sendo passado o endereço de memória para tal função. Cada linguagem, Fortran e Julia, mantém seu próprio espaço de memória que não é compartilhado durante as chamadas. Logo, algumas operações de gerenciamento de memória, como quem controla o espaço de memória atual, alocações e desalocações, são realizadas

Figura 24 – Variação no tempo de execução entre ALGENCAN via Julia e ALGENCAN via Fortran



Fonte: Autor

sempre que ocorre uma chamada.

Outro fator que adicionou latência entre as chamadas, são as conversões de tipo realizadas. Quando uma função definida em Julia é chamada pelo ALGENCAN, é realizada uma conversão automática pela linguagem Julia dos tipos passados pelo ALGENCAN. Foram tomados cuidados para diminuir ao máximo a latência, como: definir a assinatura das funções e seu retorno a fim de tomar vantagem da inferência de tipos em Julia e o uso do método `unsafe_wrap`, já mencionado, que permite envolver um *array* em Julia em torno de um endereço de memória fornecido como ponteiro sem realizar a cópia do dado.

Um desafio encarado, ao realizar testes com o CUTEst, que cabe menção, foi o cálculo das derivadas parciais sendo realizado em ordem inversa. Isto é, os vetores para a hessiana, a jacobiana e as restrições estavam apresentando os elementos de trás para frente. Após muita pesquisa, notou-se que algumas *flags* não mencionadas na documentação oficial estavam ativadas. As *flags* `lfirst` e `lvfirst` estavam marcadas como `true`, isso define a leitura das variáveis não lineares e das restrições lineares primeiro. A solução encontrada é apresentada no Código 6.1.

```
1 JuliaInterface4CUTEst.nlp = CUTEstModel(problem,lfirst=false,
    lvfirst=false)
```

Código 6.1 – Ordem de leitura das variáveis e restrições no CUTEst

7 Conclusão e Trabalhos Futuros

O presente trabalho teve como objetivo principal o desenvolvimento de uma interoperabilidade entre Julia e o ALGENCAN. O trabalho foi iniciado com a pergunta de pesquisa: *é viável do ponto de vista implementação e tempo de execução ofertar o uso do ALGENCAN na linguagem Julia?*. A partir dos resultados da implementação da interface e dos testes para a sua validação foi alcançado o objetivo principal. Com isso provou-se ser viável ofertar o uso do ALGENCAN na linguagem Julia.

Ao longo do trabalho, mostrou-se necessário ofertar o ALGENCAN em outras linguagens, pois utilizá-lo apenas em sua versão original, em Fortran, restringe o seu uso. Logo, como recomendação de trabalhos futuros está o desenvolvimento de interfaces em outras linguagens de uso científico.

Uma das limitações do trabalho é o escopo de problemas suportados pela interface. A interface desenvolvida suporta problemas de otimização não lineares e alguns lineares, desde que uma mesma restrição não esteja presente em ambas os limites do problema. O suporte a problemas que apresentem restrições em ambos os limites foge ao escopo e tempo do trabalho e, portanto, não foi incluído. Sendo assim, uma recomendação de trabalho futuro é aumentar o escopo de problemas suportados pela interface.

Os testes coletados demonstraram um aumento do tempo necessário para resolução dos problemas com a interface em Julia, fato esse justificado pelos processos envolvidos na interoperabilidade entre as linguagens, como: conversão de tipos e múltiplas chamadas.

Referências

ALAHMADI, S. et al. Performance analysis of sparse matrix-vector multiplication (spmv) on graphics processing units (gpus). *Electronics*, v. 9, 10 2020. Citado na página 20.

ANDREANI, R. et al. On augmented lagrangian methods with general lower-level constraints. *SIAM Journal on Optimization*, v. 18, p. 1286–1309, 01 2007. Citado na página 17.

ANDREANI, R. et al. On augmented lagrangian methods with general lower-level constraints. *SIAM Journal on Optimization*, v. 18, n. 4, p. 1286–1309, 2008. Disponível em: <https://doi.org/10.1137/060654797>. Citado na página 12.

ANTONIOU, W.-S. L. A. *Practical Optimization: Algorithms and Engineering Applications*. 2. ed. [S.l.]: Springer, 2021. (Texts in Computer Science). ISBN 107160841X,9781071608418. Citado na página 16.

BEZANSON, J. et al. *Julia: A Fresh Approach to Numerical Computing*. arXiv, 2014. Disponível em: <https://arxiv.org/abs/1411.1607>. Citado na página 13.

BEZANSON, J. et al. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012. Disponível em: <http://arxiv.org/abs/1209.5145>. Citado 5 vezes nas páginas 13, 21, 22, 23 e 25.

BIEGLER, L.; ZAVALA, V. Large-scale nonlinear programming using ipopt: An integrating framework for enterprise-wide dynamic optimization. *Computers Chemical Engineering*, v. 33, n. 3, p. 575–582, 2009. ISSN 0098-1354. Selected Papers from the 17th European Symposium on Computer Aided Process Engineering held in Bucharest, Romania, May 2007. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0098135408001646>. Citado na página 19.

BIRGIN, E. G.; MARTíNEZ, J. M. *Practical Augmented Lagrangian Methods for Constrained Optimization*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2014. Disponível em: <https://epubs.siam.org/doi/abs/10.1137/1.9781611973365>. Citado 4 vezes nas páginas 12, 13, 27 e 28.

BIRGIN, E. G.; MARTíNEZ, J. M. Complexity and performance of an augmented lagrangian algorithm. *Optimization Methods and Software*, Taylor Francis, v. 35, n. 5, p. 885–920, 2020. Disponível em: <https://doi.org/10.1080/10556788.2020.1746962>. Citado na página 27.

BOURQUE, P.; FAIRLEY, R. E.; SOCIETY, I. C. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3rd. ed. Washington, DC, USA: IEEE Computer Society Press, 2014. ISBN 0769551661. Citado na página 43.

BYRD, R. H.; NOCEDAL, J.; WALTZ, R. A. Knitro: An integrated package for nonlinear optimization. In: . [S.l.: s.n.], 2006. Citado 3 vezes nas páginas 13, 17 e 18.

DANG, H.-V.; SCHMIDT, B. The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus. *Procedia Computer Science*, v. 9, p. 57–66, 2012. ISSN 1877-0509. Proceedings of the International Conference on Computational Science, ICCS 2012. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050912001287>. Citado na página 19.

GOULD, N. I. M.; ORBAN, D.; TOINT, P. L. Cutest: a constrained and unconstrained testing environment with safe threads for mathematical optimization. *Computational Optimization and Applications*, v. 60, p. 545–557, 2015. Disponível em: <https://doi.org/10.1007/s10589-014-9687-3>. Acesso em: 25 jan. 2023. Citado 2 vezes nas páginas 14 e 42.

HOGG, J. D.; REID, J. K.; SCOTT, J. A. *Guidelines for the development of HSL software, 2011 version*. [S.l.]: Science and Technology Facilities Council, 2011. Citado na página 32.

JOHNSON, S. G. *The NLopt nonlinear-optimization package*. [S.l.], 2011. Disponível em: <http://ab-initio.mit.edu/nlopt>. Citado na página 12.

JULIALANG. *Calling C and Fortran Code*. 2022. Disponível em: <https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/#Calling-C-and-Fortran-Code>. Acesso em: 23 jan. 2023. Citado 3 vezes nas páginas 23, 25 e 26.

JULIALANG. *Eval of Julia Code*. 2022. Disponível em: <https://docs.julialang.org/en/v1/devdocs/eval/>. Acesso em: 23 jan. 2023. Citado 2 vezes nas páginas 21 e 22.

JULIALANG. *Types*. 2022. Disponível em: <https://docs.julialang.org/en/v1/manual/types/>. Acesso em: 23 jan. 2023. Citado na página 22.

KANZOW, C.; RAHARJA, A. B.; SCHWARTZ, A. An augmented lagrangian method for cardinality-constrained optimization problems. *Journal of Optimization Theory and Applications*, 2021. Citado na página 12.

KEMMER, T.; RJASANOW, S.; HILDEBRANDT, A. Nessie.jl – efficient and intuitive finite element and boundary element methods for nonlocal protein electrostatics in the julia language. *Journal of Computational Science*, v. 28, p. 193–203, 2018. ISSN 1877-7503. Disponível em: <https://www.sciencedirect.com/science/article/pii/S187775031730738X>. Citado 2 vezes nas páginas 21 e 22.

LACHTERMACHER, G. *Pesquisa Operacional na Tomada de Decisões*. [S.l.: s.n.], 2016. 204 p. ISBN 9788521630494. Citado na página 34.

LAWSON, C. L. et al. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, v. 5, p. 308–323, 1979. Citado na página 32.

LOBIANCO, A. *Julia Quick Syntax Reference: A Pocket Guide for Data Science Programming*. 1. ed. [S.l.]: Apress, 2019. Citado na página 23.

LUBIN, M.; DUNNING, I. Computing in operations research using julia. *INFORMS Journal on Computing*, Institute for Operations Research and the Management Sciences (INFORMS), v. 27, n. 2, p. 238–248, abr. 2015. ISSN 1526-5528. Disponível em: <http://dx.doi.org/10.1287/ijoc.2014.0623>. Citado na página 13.

MALONE, T. Interoperability in programming languages. In: . [S.l.: s.n.], 2014. Citado na página 13.

MASIERO, M. C. S. O método da função lagrangiana aumentada barreira modificada para a resolução do fluxo de potência ótimo. *Universidade Estadual Paulista, Faculdade de Engenharia*, v. 18, 08 2011. Disponível em: <http://acervodigital.unesp.br/handle/11449/87202>. Citado na página 17.

MECKLENBURG, R. *Managing Projects with GNU Make: The Power of GNU Make for Building Anything*. [S.l.]: "O'Reilly Media, Inc.", 2004. Citado na página 33.

MITTAL, D. *Particulars of Non-Linear Optimization*. Tese (Doutorado), 2015. Citado na página 12.

MOGENSEN, P. K.; RISETH, A. N. Optim: A mathematical optimization package for julia. *Journal of Open Source Software*, Open Journals, v. 3, n. 24, 2018. Citado na página 13.

MOHAMMED, T.; MEHMOOD, R. *Performance Enhancement Strategies for Sparse Matrix-Vector Multiplication (SpMV) and Iterative Linear Solvers*. 2022. Citado na página 19.

NOCEDAL, J.; WRIGHT, S. J. *Numerical optimization / Jorge Nocedal, Stephen J. Wright*. 2nd ed.. ed. New York: Springer, 2006. (Springer series in operations research and financial engineering). ISBN 0387303030. Citado na página 12.

ORBAN, D.; SIQUEIRA, A. S.; contributors. *NLPModels.jl: Data Structures for Optimization Models*. 2020. <https://github.com/JuliaSmoothOptimizers/NLPModels.jl>. Citado na página 42.

PRASETYA, K. D.; SUHARJITO; PRATAMA, D. Effectiveness analysis of distributed scrum model compared to waterfall approach in third-party application development. *Procedia Computer Science*, v. 179, p. 103–111, 2021. ISSN 1877-0509. 5th International Conference on Computer Science and Computational Intelligence 2020. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050920324558>. Citado na página 39.

PRESSMAN, R.; MAXIM, B. *Software Engineering: A Practitioner's Approach, Ninth Edition*. Ji xie gong ye chu ban she, 2021. (). ISBN 9787111690726. Disponível em: <https://books.google.com.br/books?id=hTTGzgEACAAJ>. Citado na página 39.

RANOCHA, H. et al. Adaptive numerical simulations with trixi.jl: A case study of julia for scientific computing. *Proceedings of the JuliaCon Conferences*, The Open Journal, v. 1, n. 1, p. 77, 2022. Disponível em: <https://doi.org/10.21105/jcon.00077>. Citado na página 13.

SANTOS, A. B. e Caroline Guimarães e Leandro Freitas e L. Algoritmos de otimização aplicados à solução de sistemas estruturais não-lineares sem e com restrições: uma abordagem utilizando os métodos da penalidade e do lagrangiano aumentado. *Exacta*, v. 8, n. 3, p. 345–361, 2011. ISSN 1983-9308. Disponível em: <https://link.springer.com/book/10.1007/978-1-0716-0843-2>. Citado na página 16.

STALLMAN, R. M. *Using the gnu compiler collection: a gnu manual for gcc version 4.3. 3*. [S.l.]: CreateSpace, 2009. Citado 2 vezes nas páginas 25 e 26.

THESING, T.; FELDMANN, C.; BURCHARDT, M. Agile versus waterfall project management: Decision model for selecting the appropriate approach to a project. *Procedia Computer Science*, v. 181, p. 746–756, 2021. ISSN 1877-0509. CENTERIS 2020 - International Conference on ENTERprise Information Systems / ProjMAN 2020 - International Conference on Project MANagement / HCist 2020 - International Conference on Health and Social Care Information Systems and Technologies 2020, CENTERIS/ProjMAN/HCist 2020. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050921002702>. Citado 2 vezes nas páginas 39 e 40.

VENTER, G. Review of optimization techniques. In: _____. *Encyclopedia of Aerospace Engineering*. John Wiley Sons, Ltd, 2010. ISBN 9780470686652. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470686652.eae495>. Citado na página 12.

VERDUGO, F.; BADIA, S. The software design of gridap: A finite element package based on the julia JIT compiler. *Computer Physics Communications*, Elsevier BV, v. 276, p. 108341, jul 2022. Disponível em: <https://doi.org/10.1016/j.cpc.2022.108341>. Citado na página 22.

WäCHTER, A.; BIEGLER, L. T. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, v. 106, 2006. Disponível em: <https://doi.org/10.1007/s10107-004-0559-y>. Citado na página 13.

Apêndices

APÊNDICE A – Interface com o CUTEst em Julia

```
module JuliaInterface4CUTEst
  using NLPModels, CUTEst
  export MyDataPtr,evalf!,evalc!,evalg!,evalj!,evalhl!,
   problem_params, nlp

  mutable struct MyDataPtr
    counters::NTuple{5, Int32}

    MyDataPtr(counters) = new(counters)
  end

  nlp = ""

  function problem_params()::Tuple
    # Number of variables
    n::Int32 = nlp.meta.nvar

    # Initial guess and bound constraints

    x = Vector{Float64}(nlp.meta.x0)

    # Stores the objective function
    f::Float64 = 0.0

    lind = Vector{Int32}(zeros(n))
    lbnd::Vector{Float64} = [-1.0e20 for _ in 1:n]

    uind = Vector{Int32}(zeros(n))
    ubnd::Vector{Float64} = [1.0e20 for _ in 1:n]

    lvar = nlp.meta.lvar
    uvar = nlp.meta.uvar
    for k in 1:length(lvar)
      if (lvar[k] != -Inf)
        lind[k] = 1
```

```
35          lbnd[k] = lvar[k]
36        end
37
38        if (uvar[k] != Inf)
39          uind[k] = 1
40          ubnd[k] = uvar[k]
41        end
42      end
43
44      # Number of equality (m) and inequality (p) constraints
45      m = length(nlp.meta.jfix)
46      p = length(nlp.meta.jlow) + length(nlp.meta.jupp) + length(
    nlp.meta.jrng)
47
48      # Initial guess for the Lagrange multipliers
49
50      lambda = Vector{Float64}(zeros(m+p))
51
52      (*@ \pagebreak @*)
53      # Number of entries in the Jacobian of the constraints
54
55      jnnzmax::Int32 = 2 * nlp.meta.nnzj
56
57      # This should be the number of entries in the Hessian of the
58      # Lagrangian. But, in fact, some extra space is need (to
    store the
59      # Hessian of the Augmented Lagrangian, whose size is hard to
60      # predict, and/or to store the Jacobian of the KKT system).
    Thus,
61      # declare it as large as possible.
62
63      hlnnzmax::Int32 = typemax(Int32)    10
64
65      # Feasibility, complementarity, and optimality tolerances
66
67      epsfeas::Float64  = 1.0e-08
68      epscompl::Float64 = 1.0e-08
69      epsopt::Float64   = 1.0e-08
70
71      maxoutit::Int32 = 50
72
```

```
73     # rhoauto means that Algencan will automatically set the
       initial
74     # value of the penalty parameter. If you set rhoauto = .false
       . then
75     # you must set rhoini below with a meaningful value.
76
77     rhoauto::Int32 = 1
78     rhoini::Float64 = 0.0
79
80     if !Bool(rhoauto)
81       rhoini = 1.0e-08
82     end
83
84     # scale = .true. means that you allow Algencan to
       automatically
85     # scale the constraints. In any case, the feasibility
       tolerance
86     # (epsfeas) will be always satisfied by the UNSCALED original
87     # constraints.
88     scale::Int32 = 0
89
90     # extallowed = .true. means that you allow Gencan (the active
       -set
91     # method used by Algencan to solve the bound-constrained
92     # subproblems) to perform extrapolations. This strategy may
       use
93     # extra evaluations of the objective function and the
       constraints
94     # per iterations; but it uses to provide overal savings. You
       should
95     # test both choices for the problem at hand.
96     extallowed::Int32 = 1
97
98     # corrin = .true. means that you allow the inertia of the
99     # Jacobian of the KKT system to be corrected during the
       acceleration
100    # process. You should test both choices for the problem at
       hand.
101    corrin::Int32 = 0
102
103    inform::Int32 = 0
104
```

```
105     pdata::MyDataPtr = MyDataPtr((0,0,0,0,0))
106
107     return x,n,f,lind,lbnd,uind,ubnd,m,p,lambda,jnnzmax,hlnnzmax,
    epsfeas,epscompl,epsopt,
108            rhoauto,rhoini,scale,extallowed,corrin,inform,maxoutit,
    pdata
109   end
110
111   function evalf!(n::Int32,x::Ptr{Float64},f::Ptr{Float64},inform
    ::Int32,pdataptr::Ptr{MyDataPtr}=nothing)::Nothing
112     # I should define an equivalent call to c_f_pointer(pdataptr,
    pdata) and an equivalent structure to pdata and pdataptr
113     x_wrap::Vector{Float64} = unsafe_wrap(Array, x, n)
114     f_wrap::Vector{Float64} = unsafe_wrap(Array, f, 1)
115
116     temp::Float64 = convert(Float64, obj(nlp, x_wrap))
117
118     unsafe_store!(f, temp)
119
120     nothing
121   end
122
123   function evalg!(n::Int32,x::Ptr{Float64},g::Ptr{Float64},inform
    ::Int32,pdataptr::Ptr{MyDataPtr}=nothing)::Nothing
124     # I should define an equivalent call to c_f_pointer(pdataptr,
    pdata) and an equivalent structure to pdata and pdataptr
125     x_wrap::Vector{Float64} = unsafe_wrap(Array, x, n)
126     g_wrap::Vector{Float64} = unsafe_wrap(Array, g, n)
127
128     df = (x_arg::Vector{Float64}) -> convert(Vector{Float64},grad
    (nlp, x_arg))
129
130     temp::Vector{Float64} = df(x_wrap)
131
132     for j in 1:n
133       g_wrap[j] = temp[j]
134     end
135
136     nothing
137   end
138
139   function evalc!(
```

```
140     n::Int32,x::Ptr{Float64},m::Int32,p::Int32,c::Ptr{Float64},
    inform::Int32,pdataptr::Ptr{MyDataPtr}=nothing
141     )::Nothing
142     # I should define an equivalent call to c_f_pointer(pdataptr,
    pdata) and an equivalent structure to pdata and pdataptr
143     if ((m+p) == 0)
144       return nothing
145     end
146
147     x_wrap::Vector{Float64} = unsafe_wrap(Array, x, n)
148     c_wrap::Vector{Float64} = unsafe_wrap(Array, c, (m+p))
149
150     temp::Vector{Float64} = convert(Vector{Float64}, cons(nlp,
    x_wrap))
151
152     low_constr_idx = nlp.meta.jlow
153     for j in 1:(m+p)
154       if j in low_constr_idx
155         c_wrap[j] = temp[j]*-1.0
156       else
157         c_wrap[j] = temp[j]
158       end
159     end
160
161     nothing
162   end
163
164   function evalj!(n::Int32,x::Ptr{Float64},m::Int32,p::Int32,ind
    ::Ptr{Int32},
165     sorted::Ptr{Int32},jsta::Ptr{Int32},jlen::Ptr{Int32},lim::
    Int32,
166     jvar::Ptr{Int32},jval::Ptr{Float64},inform::Int32,pdataptr::
    Ptr{MyDataPtr}=nothing
167     )::Nothing
168       # I should define an equivalent call to c_f_pointer(
    pdataptr,pdata) and an equivalent structure to pdata and
    pdataptr
169     if ((m+p) == 0)
170       return nothing
171     end
172
173     x_wrap = unsafe_wrap(Array, x, n)
```

```
174     ind_wrap = unsafe_wrap(Array, ind, m+p)
175     sorted_wrap = unsafe_wrap(Array, sorted, m+p)
176     jsta_wrap = unsafe_wrap(Array, jsta, m+p)
177     jlen_wrap = unsafe_wrap(Array, jlen, m+p)
178     jvar_wrap = unsafe_wrap(Array, jvar, lim)
179     jval_wrap = unsafe_wrap(Array, jval, lim)
180
181     csc2csr(nlp,x_wrap,jval_wrap,jvar_wrap,jsta_wrap,jlen_wrap)
182
183     for i in 1:(m+p)
184       sorted_wrap[i] = 0
185       if (ind_wrap[i] != 0)
186         if ( lim < n )
187           error_code::Int32 = -94
188           unsafe_store!(inform, error_code)
189           return
190         end
191
192       end
193     end
194
195     # I need to multiply only the lines of inequality constraints
      in order to convert the operator to that accepted by ALGENCAN
    (<=)
196     low_constr_idx = nlp.meta.jlow
197     if (length(low_constr_idx) > 0)
198       for i in (m*n)+1:length(jval_wrap)
199         jval_wrap[i] = jval_wrap[i] * -1.0
200       end
201     end
202
203     nothing
204   end
205
206   function evalhl!(
207     n::Int32,x::Ptr{Float64},m::Int32,p::Int32,lambda::Ptr{
     Float64},lim::Int32,
208     inclf::Int32,hlnnz::Ptr{Int32},hlrow::Ptr{Int32},hlcol::Ptr{
     Int32},hlval::Ptr{Float64},
209     inform::Ptr{Int32},pdataptr::Ptr{MyDataPtr}=nothing
210     )::Nothing
```

```
211     # I should define an equivalent call to c_f_pointer(pdataptr,
    pdata) and an equivalent structure to pdata and pdataptr
212     x_wrap = unsafe_wrap(Array, x, n)
213     hlcol_wrap = unsafe_wrap(Array, hlcol, lim)
214     hlval_wrap = unsafe_wrap(Array, hlval, lim)
215     lambda_wrap = unsafe_wrap(Array, lambda, m+p)
216     inform_wrap = unsafe_wrap(Array, inform, 1)
217     error_code::Int32 = -95
218
219     hlnnz_temp = 0
220
221     # If .not. inclf then the Hessian of the objective function
    must not be included
222
223     if (Bool(inclf))
224       if ( hlnnz_temp + 2 > lim )
225         unsafe_store!(inform, error_code)
226         return
227       end
228
229       low_constr_idx = nlp.meta.jlow
230       if (length(low_constr_idx) > 0)
231         for i in low_constr_idx
232           lambda_wrap[i] = lambda_wrap[i] * -1.0
233         end
234       end
235
236       hlval_temp = hess_coord(nlp, x_wrap, lambda_wrap)
237       hlrow_temp, hlcol_temp = hess_structure(nlp)
238
239       for val in 1:length(hlval_temp)
240         unsafe_store!(hlval, hlval_temp[val], val)
241       end
242
243       for row in 1:length(hlrow_temp)
244         unsafe_store!(hlcol, hlrow_temp[row], row)
245       end
246
247       for col in 1:length(hlcol_temp)
248         unsafe_store!(hlrow, hlcol_temp[col], col)
249       end
250
```

```
251       hlnnz_temp = nlp.meta.nnzh
252       unsafe_store!(hlnnz, hlnnz_temp)
253     end
254
255     # Note that entries of the Hessian of the Lagrangian can be
256     # repeated. If this is case, them sum of repeated entrances
    is
257     # considered. This feature simplifies the construction of the
258     # Hessian of the Lagrangian.
259
260     if (hlnnz_temp + 1 > lim)
261       unsafe_store!(inform, error_code)
262       return
263     end
264
265     nothing
266   end
267
268   function csc2csr(nlp_obj::CUTEstModel,x_wrap::Vector{Float64},
     val2::Vector{Float64},col::Vector{Int32},ind2::Vector{Int32},
     line_len::Vector{Int32})
269     jacobian = jac(nlp_obj, x_wrap)
270     m::Int64 = jacobian.m
271     n::Int64 = jacobian.n
272     ind1::Vector{Int64} = jacobian.colptr
273     val1::Vector{Float64} = jacobian.nzval
274     row::Vector{Int64} = jacobian.rowval
275     nnz::Int64 = length(jacobian.nzval)
276     cnt = Vector{Int64}(zeros(m))
277
278     # Cross row to check how many non zero elements there are in
    each line
279     for i in 1:nnz
280       cnt[row[i]] = cnt[row[i]] + 1
281     end
282
283     # Fill ind2 with indexes indicating where starts each line
    according to val1
284     ind2[1] = 1
285     for i in 2:m
286       ind2[i] = ind2[i-1] + cnt[i-1]
287
```

```
288       # Copy num of elements per line from cnt to jlen
289       line_len[i-1] = cnt[i-1]
290
291       # Copy ind2 to cnt
292       cnt[i-1] = ind2[i-1]
293     end
294     line_len[m] = cnt[m]
295     cnt[m] = ind2[m]
296
297     # Fill col and val2 using cnt and row as guides
298     for j in 1:n-1
299       for i in ind1[j]:ind1[j+1]-1
300         col[cnt[row[i]]] = j
301         val2[cnt[row[i]]] = val1[i]
302         cnt[row[i]] = cnt[row[i]] + 1
303       end
304     end
305
306     # Cross the last column
307     for j in ind1[n]:nnz
308       col[cnt[row[j]]] = n
309       val2[cnt[row[j]]] = val1[j]
310       cnt[row[j]] = cnt[row[j]] + 1
311     end
312   end
313 end
```

Código A.1 – Interface com CUTEst em Julia

APÊNDICE B – Script para guiar os testes em Julia

```
"Defines a set of problems and run tests with them over ALGENCAN shared library."
module JuliaInterface4CUTEstTest
  using Test, NLPModels, CUTEst
  include("./asserts/julia_interface_4_CUTEst.jl")

  using .JuliaInterface4CUTEst

  include("../algencan_interface.jl")

  export run_tests

  """
    Executes tests over Julia interface with ALGENCAN from a defined set of 17 CUTEst problems.
  """
  function run_tests()::Nothing
    problem_set::Vector{String} = ["ANTWERP","HATFLDC","ORTHREGB","KSIP","BQPGABIM","OPTCNTRL","SANTALS","ERRINRSM","BQPGASIM","GOULDQP1","ACOPP14","ERRINROS","HATFLDGLS","TOINTQOR","3PK","METHANB8LS"]
    file = open("julia_interface_4_CUTEst_results.txt","w")

    for problem in problem_set

      @testset "Check algencan_interface behaviour over $problem problem" begin
        JuliaInterface4CUTEst.nlp = CUTEstModel(problem,lfirst=false,lvfirst=false)

        time_spent::Float64 = @elapsed begin
          x::Vector{Float64} = run_algencan()
        end

        f::Float64 = obj(JuliaInterface4CUTEst.nlp, x)
```

```
29
30          println("========================RESULTS
    ========================")
31          println("f = $f")
32          println("Time spent(seconds) = $time_spent")
33
34          write(file,"$problem\t\t\t\t$time_spent\t\t\t\t$f\n")
35
36          finalize(JuliaInterface4CUTEst.nlp)
37        end
38      end
39
40      close(file)
41    end
42
43    run_tests()
44  end
```

Código B.1 – Script para guiar os testes em Julia