Projeto de Cálculo Numérico: Problema do Caixeiro Viajante em Julia

Iann Takami Singo
Júlio César Fagundes
Luiza Gabriela da Silva
Victor Gabriel Zerger

Resumo: Este projeto tem o objetivo de apresentar um algoritmo de resolução para o Problema do Caixeiro Viajante, o qual consiste em definir a menor rota entre determinadas cidades, sem passar por uma mais de uma vez, saindo da cidade de Curitiba, e voltando nela. O algoritmo foi feito na linguagem Julia e tem como base o modelo de Lin-Kernighan, publicado em 1973, que serve perfeitamente para nosso caso, pois não iremos ultrapassar de cinquenta cidades.

***Palavras-chave*:** Caixeiro Viajante. Julia.

1 Por quê do tema?

O Problema do Caixeiro Viajante foi escolhido por alguns motivos. O primeiro deles foi devida a situação de pandemia que estamos vivendo, onde as compras onlines aumentaram muito em 2020, uma vez que todos deveriam ter ficado em casa em isolamento social. Outra consideração é que ideia do problema é facilitar às entregas das mercadorias de transportadoras, uma vez que dependendo da rota escolhida, o custo e a distância serão maiores, e dada a economia do país, gastar menos é importante para qualquer pessoa, principalmente para as empresas. O último motivo é sobre a eficiência dos modelos já propostos, uma vez que não é fácil acertar com uma precisão de 100% a distância, que dirá uma aproximação para os custos. Com isso, nosso projeto é baseado nas ideias de Lin-Kernighan, onde o número máximo de pontos ou, cidades, é de 50, e possui um acerto muito bom. Este modelo é suficiente para nós pois, nos nossos testes usamos apenas 5 e funcionou muito bem, acreditamos que funcione para mais cidades também.

2 O Problema

O Problema do Caixeiro Viajante é interessante pois não há um modelo pronto e preciso que funcione para n casos. O algortimo de Lin-Kernighan, publicado em 1973, é o que possui mais sucesso, porém, como mencionado anteriormente, só funciona em até 50 casos, para menores valores a porcentagem de acerto cai drásticamente. Mesmo assim, o algoritmo é utilizado para a obtenção de rotas, no nosso caso em particular, rotas de entrega.

O que torna esse problema complicado são os critérios de funcionamento. Um deles é que uma cidade não pode estar na rota mais que uma vez, com exceção da cidade de origem, por isso, há várias condicionais a serem aplicadas com o intuito de manter a ordem do problema. No total temos 5 delas, e terá uma subseção específica para falar detalhadamente sobre casa uma.

As vantagens em se obter a menor rota possível estão vinculadas em sua maioria ao dinheiro. Rotas maiores gastam mais combustível, as pessoas que fazem as entregas ficam mais tempo na estrada e, consequentemente devem ganhar a mais pelo seu tempo de serviço. Além disso, temos o perigo contante das rodovias que põe a vida desses trabalhadores em risco todos os dias, portanto a ideia de traçar a menor rota é muito boa tanto para os funcionários, como para as empresas.

3 Metodologia

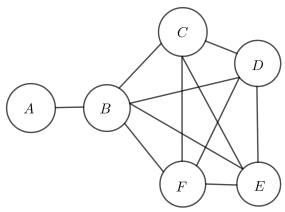
Para conectar a ideia do caixeiro viajante à rotas reais de entregas foi preciso percorrer um caminho de pesquisas e leituras sobre o tema. Com isso, artigos e sites foram lidos e muitos vídeos assistidos. Quando

pensamos na situação de rotas reais temos que levar em consideração muitos aspectos, o principal deles: não há estradas perfeitamente retas, com a menor distância possível, que ligam uma cidade a outra. Isto foi fundamental para enterdermos que precisaríamos começar com coisas palpáveis e perder um pouco da ideia do "real". Por isso, as cidades agora podiam ser vistas como pontos de uma malha bidimensional e as distâncias que precisamos são as ideais, ou seja, retas com a menor rota entre elas. Aqui cabe um comentário muito relevante, se conseguissemos criar o código que funcionasse para esse tipo de malha, implementar a distância da Geometria Não Euclidiana nele seria um grande passo do grupo para o projeto ir ganhando dificuldade.

Todo esse sistema num primeiro momento se encaixou perfeitamente na teoria de grafos, porém, para satisfazer nossas prioridades precisávamos estudar todas as condicionais do problema baseadas no método que escolhemos (Lin-Kernighan) para enfim, conseguir implementar isto em Julia. A ideia de começar pequeno e utilizar uma malha bidimensional facilita para calcular a distância entre dois pontos, pois a fórmula é bem conhecida e dada por: $d = \sqrt{(x - x_o)^2 + (y - y_0)^2}$; onde d é a distância, (x,y) e (x_o, y_o) são as coordenadas dos pontos que queremos calcular.

3.1 Grafos

O primeiro passo foi compreender alguns termos da Teoria de Grafos, pois podemos considerar as cidades como vértices ou nós. Isso foi importante para estudar como funcionaria as arestas (distâncias) entre os vértices, e que a direção tinha uma relevância muito grande para nosso projeto. Na figura temos um exemplo de grafo com arestas conectando todos os vértices, mas para o nosso caso não podemos utilizar todas elas, caso contrário estaríamos passando por uma cidade mais de uma vez.



Há uma fórmula que dado o número de vértices, podemos calcular quantas arestas o grafo completo, deveria ter, que é:

$$K_n = \frac{n \cdot (n-1)}{2} \tag{1}$$

Onde K_n são as arestas e n o número de nós. Se formos calcular com n = 6, como no nosso exemplo, o grafo deveria ter 35 arestas, que é um número muito maior do que realmente tem. Isso significa que um vértice não está fechando o grafo, o que é visível, dado que o nó A não possui ligação com mais nenhum outro. Este tipo de problema será resolvido com uma restrição no próximo tópico. Um ponto importante a ser mencionado é que no desenvolvimento do cógido não usamos a Teoria em si, com a notação colocada acima e todas as preocupações e conceitos dos grafos. Mesmo não usando a Teoria em si, ela foi importante para compreendermos sobre as condicionais e para tentarmos criar ao final do código um gráfico onde mostrassem as cidades e a rota mais curta, porém não conseguimos.

3.2 Condicionais

As condicionais nada mais são do que restrições que adequam o Problema do Caixeiro Viajante ao nosso projeto, que é focado em rotas de entrega. Então, perguntas como: *pode passar mais de uma vez na mesma cidade?;* é respondida e resolvida com as equações que iremos colocar neste tópico.

Primeiramente devemos considerar nossa distância como d_{ij}, que significa *"distância do nó i para o nó j"*, onde $i, j \in \mathbb{N}$. Com isso, a formulação matemática para este problema é dada por:

$$\textit{Min} = \sum_{i=1}^{n} d_{ij} x_{ij} \tag{2}$$

Onde *Min* é uma função minimizar; d_{ij} a distância entre i e j; e x_{ij} representa uma variável.

Temos agora as condicionais:

$$\sum_{j=1}^{n} x_{ij} = 1,\ i = 1, ..., n;\ i \neq j \tag{3}$$

$$\sum_{i=1}^{n} x_{ij} = 1,\ j = 1, ..., n;\ j \neq i \tag{4}$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1,\ S \subset \{1, ..., n\},\ 2 \leq |S| \leq n - 2 \tag{5}$$

$$x_{ij} = 0 \text{ ou } 1;\ i, j \in \{1, ..., n\} \tag{6}$$

$$x[i, j] + x[j, i] <= 1 \tag{7}$$

As equações (3) e (4) nos mostram que existe uma aresta de chegada e partida de i. A quinta nos garante que não existam rotas incompletas, ou seja, teremos que tomar cuidado caso aconteça o problema dado no exemplo dos grafos, onde A não está ligado com todos os outros nós. Já a equação (6) nos mostra que a variável x_{ij} pode assumir dois valores: 1 indica uma relação de ordem, saiu do ponto i e foi para o j; ou 0, que significa que não aconteceu o 1. A equação (7) foi adicionada pois na matriz distância os elementos ij de x, são os mesmo de ji. Essa condicional impede que você pegue duas vezes o mesmo local. Nossa maior dificuldade do projeto foi justamente o "coração"do Problema, que é implementar essas condições no código em Julia.

4 Código

Entendido o Problema do Caixeiro Viajante e suas especificações, começamos a montar o código, e esta subseção será dedicada aos perrengues e acertos que tivemos neste processo.O primeiro empecilho foi como conseguir a latitude e longitude das cidades que o usuário gostaria de ter, imaginando que este código será usado por empresas que façam entregas, os pontos seriam diferentes em todos os casos e por isso, tentamos fazê-lo de um modo mais geral possível.

Nossos professores deram a ideia de usar pacotes específicos denominados: *CSV e DataFrames*, onde CSV.write escreve um arquivo CSV e salva no computador. Primeiramente o local do arquivo é importado para que a função consiga ler, e quando isto acontecer, será "lido"no tipo *dataframe* , ou seja, uma tabela em Excel ou .csv, e isso é transformado para que a gente consiga trabalhar com os dados do arquivo.

Para nós, ficou mais fácil utilizar as tabelas em *.csv*, porém tivemos que ter bastante cuidado com o que as linhas e colunas estavam dando de informações, e como acessar corretamente as entradas que queríamos em Julia. Começamos com algo parecido com:

```
1. using DataFrames
2. df = DataFrame(Name =
             ["Cidade 1","Cidade 2","Cidade 3","Cidade 4","Cidade 5"],
             Latitude = [2212312,43234234,8153453,5234232,21231237],
             Longitude = [30000,45000,60000,50000,55000]
             )

3. using CSV
4. CSV.write("C:\\Users\\Julio\\Desktop\\ArquivoCidades.csv", df)
5.####Este é o comando que escreve o arquivo em formato .csv e salva no computador
```

Este início do nosso código então, é voltado para termos em mãos as cidades e suas respectivas latitudes e longitudes. O arquivo com nome *export_df.csv* que está em nossas contas do GitHub são cidades para teste (mais precisamente foram 5 cidades), como se fosse um usuário ou cliente enviando pra gente. Vale ressaltar que o código foi feito de maneira geral, não com os dados da tabela do arquivo, só usamos ela para referência, ou seja, ver se o código estava funcionando.

Aqui está a tabela usada no arquivo *export_df.csv*:

Cidade	Latitude	Longitude
Cidade 1	-16.75	-49.43
Cidade 2	-18.48	-47.4
Cidade 3	-16.2	-48.7
Cidade 4	-19.16	-45.44
Cidade 5	-1.71	-48.88

4.1 Distância

Depois de resolvida a questão de como conseguir as informações das cidades, o próximo problema foi qual fórmula da distância usar, mesmo combinado que iríamos fazer com a fórmula bidimensional, um colega do grupo começou a procurar e entender mais sobre a fórmula da Geometria Não Euclidiana. Quisemos colocar esta subseção para falar que conseguimos utilizar a fórmula! Porém, ficaremos no "vai e vem"de códigos, pois estávamos fazendo dois simultaneamente para ver qual funcionava melhor. Chamaremos de **Código 1** o da fórmula da distância bidimensional, e **Código 2** o da distância não Euclidiana.

A fórmula encontrada pelo colega e como foi tentado implementar no código 2, primeiramente foi:

```
 1. using LinearAlgebra
 2. function nota(a::Matrix{Float64})
 3.  (m,n)=size(a)
 4.   s = rand(m,m)
 5.   for i = 1:m
 6.       for j= 1:m
 7.           x = 6371*acos((cos((90-a[i,1])/180*pi))*cos((90-a[j,1])/180*pi)
             +sin((90-a[i,1])/180*pi)*sin((90-a[j,1])*pi/180)
             *cos((a[i,2]-a[j,2])*pi/180))*1.15
 8.           s[i,j] = x
 9.       end
10.   end
11. return s
12. end
```

Como tivemos problema com a exatidão do programa Julia, já que o mesmo resultou em um acos(1.0000002), começamos a ter que buscar outras formas de encontrar distância entre dois pontos no globo de forma mais aproximada, assim veio a ideia de utilizar regressão linear, a qual utilizamos a distância de pontos da latitude e longitude das cidades. Então, nosso x seria a distância entre dois pontos e nosso y seria o real valor da distância em quilômetros que pegamos no Google Maps. A fórmula de regressão é $m\theta(x) = \theta_1 x + \theta_0$ e encontramos o valor de θ_1 pelo somatório:

$$\frac{\frac{\sum_{i=1}^{n} x_i}{n} \cdot \frac{\sum_{i=1}^{n} y_i}{n} - \frac{\sum_{i=1}^{n} x_i y_i}{n}}{\frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n} - \frac{\sum_{i=1}^{n} x_i^2}{n}}$$

Onde n é o número de cidades; i = 1,...,n e θ_1 é 129.379. Após encontrar o valor anterior, substituimos para encontrar o valor de θ_0 e conseguimos a fórmula

$$m(x) = 129.379x - 34.8839.$$

Logo, esta fórmula aproxima o máximo possível o x (distância Euclidiana) para km, e utilizamos ela na fórmula da distância Não Euclidiana, apresentada anteriormente.

A junção das partes das distâncias, tanto bidimensional como não euclidiana, com a que vem a seguir sobre as condicionais, foi muito difícil em ambos os códigos.

4.2 Condicionais

O coração do nosso projeto são as condicionais de (3) a (6) apresentadas anteriormente. Essas restrições foram inspiradas nas ideias de Lin-Kernighan e a mais difícil de implementar foi a (5), que elimina as subrotas, ou rotas que não são boas para os códigos. Temos que lembrar que na variável x_{ij}, a cidade j é a de chegada, e ela não pode ser passada mais de uma vez, o que dificultou muito o desenvolvimento do código.

Tivemos que baixar e aprender a usar pacotes específicos como: PrettyTables; Clp; JuMP e GLPK. Suas funções são: imprimir dados em matrizes em um formato legível; uma interface para o solver de programação linear COIN-OR; pacote de modelagem matemática para otimização linear e não linear; um invólucro para a biblioteca do Kit de Programação Linear GNU, respectivamente.

No final, descobrimos que restrições são adicionadas a um modelo JuMP com o comando *@constraint*, e com isso nossa condicional 5 ficou menos pior de entender e aplicar.

```
1. for i = 1:m, j = 1:m
2.     @constraint(f, x[i,j]+x[j,i] <= 1)
3. end
```

Explicando: este comando impede de passarmos pela mesma cidade duas vezes, ou seja impede que a matriz s não pegue mesmo valores na matriz distância, porque os valores de x[i,j] e x[j,i] são os mesmo na matriz s. Qual a matriz distância e o que é essa matriz s, colocaremos o link para o código depois das conclusões.

5 Conclusões

Tivemos muitos erros ao longo do desenvolvimento deste projeto. O primeiro deles foi falta de comunicação, enquanto alguns faziam a ideia inicial de começar simples e depois ir aumentando a dificuldade, outros estavam fazendo o mais complicado primeiro, ao final tudo se complementou, porém tivemos este problema no início. Outra grande dificuldade foi aprender alguns conceitos e comandos de determinados pacotes em um período relativamente curto, felizmente tivemos uma tentativa de código que funcionou como esperávamos, e contraditóriamente foi o que tinham as dificuldades maiores.

No código 1, a cada tentativa de restrição das subrotas dadas pela equação (5), acabávamos criando mais uma especificação, ou seja, conforme o código ia sendo criado ele só funcionava para determinados casos e não para todos como gostaríamos. A cada tentativa o grupo ficava desestabilizado, pois encontrávamos contra-exemplos

sempre, e com isso a ideia de entregar um programa que não estava funcionando como gostaríamos abalou as nossas estruturas emocionais. Este código estará em nossos respectivos GitHubs, pois deu muito trabalho para fazê-lo e foi com o início dele que o outro código deu certo.

No código 2 utilizamos as cidades teste, que foram só 5, e fizemos o "teste de mesa", o qual funcionou perfeitamente, infelizmente para mais cidades não sabemos a eficácia, mas confiamos no projeto. Acreditamos também que devida a excelência do método que escolhemos (Lin-Kernighan) ele funcione em até 50 cidades. Como já foi dito anteriormente o início deste código foi copiado do primeiro, uma vez que independente de qual fórmula da distância usássemos, era preciso ter uma matriz das cidades e suas respectivas latitudes e longitudes.

Conseguir calcular a distância com a Geometria Não Euclidiana, calcular os custos e printar a ordem certa das cidades foram nossos maiores acertos em relação no desenvolvimento do código. Como grupo, tivemos nossos momentos de desencontro mas que ao final se encaixaram perfeitamente. Ninguém fez mais ou menos que o outro, teve um equilíbrio muito bom.

5.1 GitHub

Segue o link para o GitHub dos respectivos criadores deste projeto, nele temos os dois códigos citados neste relatório com nomes: Código 1(malha bidimensional) e Código 2(distância não Euclidiana); este relatório e o slide para a apresentação do dia 14/12/2020.

Link GitHub Iann Takami Singo: <https://github.com/iannts/Projeto-1>

Link GitHub Júlio César Fagundes: <https://github.com/JulioCFagundes/Projeto1>

Link GitHub Luiza Gabriela da Silva: <https://github.com/Lu1zaGabriela/Projeto-1>

Link GitHub Victor Gabriel Zerger: <https://github.com/Victor-Zerger/Projeto-1>

6 Referências

[1] AZEVEDO, Anibal. Aula 31 - Módulo 4.2 - Formulação Matemática do Problema do Caixeiro Viajante. Youtube. Disponível em: <https://www.youtube.com/watch?v=GAiiE8rMzM4> Acesso em: 30 de nov. 2020.

[2] UNIVESP. Pesquisa Operacional II - Aula 07 - O problema do caixeiro viajante. Youtube. Disponível em: <https://www.youtube.com/watch?v=Doy6cBjb8uw> Acesso em: 1 de dez. 2020.

[3] REIS, Jorge. UFU Exercício 106 Problema do Caixeiro Viajante utilizando o Solver. Youtube. Disponível em: <https://www.youtube.com/watch?v=PfQ8-yZXXc4> Acesso em: 1 de dez. 2020.

[4] JCharisTech & J-Secur1ty. Julia Programming Tutorials:Plotting Data. Youtube. Disponível em: <https://www.youtube.com/watch?v=Zym7Qe2cjAs> Acesso em: 7 de dez. 2020.

[5] Abel Siqueira; Tutorial de Julia em Português - Pacote Plots.jl. Youtube. Disponível em: <https://www.youtube.com/watch?v=bowHvSlam7M&list=PLOOY0eChA1uyjKRQDeR4-LfsObNqslIUh&index=1> Acesso em: 7 de dez. 2020.

[6] Abel Siqueira; Tutoriais de Julia em Português - JuMP.jl parte 1 - O básico. Youtube. Disponível em: <https://www.youtube.com/watch?v=v1hJsmMqpvw&list=PLOOY0eChA1uyjKRQDeR4-LfsObNqslIUh&index=8> Acesso em: 7 de dez. de 2020.

[7] Abel Siqueira; Tutoriais de Julia em Português - JuMP.jl parte 2 - Vendendo sua arte na praia. Youtube. Disponível em: <https://www.youtube.com/watch?v=oz6S5vTE2tM&list=PLOOY0eChA1uyjKRQDeR4-LfsObNqslIUh&index=9> Acesso em: 7 de dez. de 2020.

[8] Abel Siqueira; Tutoriais de Julia em Português - JuMP.jl parte 3 - Um exemplo grande e complicado. Youtube. Disponível em: <https://www.youtube.com/watch?v=377MmkGdm2Q&list=PLOOY0eChA1uyjKRQDeR4-LfsObNqslIUh&index=10> Acesso em: 8 de dez. de 2020.

[9] Abel Siqueira; Tutoriais de Julia em Português - JuMP.jl parte 4 - Modelo não linear - regressão logística. Youtube. Disponível em: <https://www.youtube.com/watch?v=1p3kihLbFHU&list=PLOOY0eChA1uyjKRQDeR4-LfsObNqslIUh&index=11> Acesso em: 8 de dez. de 2020.

[10] Abel Siqueira; Tutoriais de Julia em Português - JuMP.jl parte 5 - Não linear inteira mista. Youtube. Disponível em: <https://www.youtube.com/watch?v=wCKodtczy1I&list=PLOOY0eChA1uyjKRQDeR4-LfsObNqslIUh&index=12> Acesso em: 8 de dez. de 2020.

[11] Matemática e Áreas correlatas com Prof. Lopo; Caixeiro Viajante-Problema de Grafos resolvido com Excel-Solver. Youtube. Disponível em: <https://www.youtube.com/watch?v=6JaGgnAOob> Acesso em: 30 de nov. de 2020.

[12] CASTELLUCCI, P. (2017). JULIA E JuMP: NOVAS FERRAMENTAS PARA PROGRAMAÇÃO MATEMÁTICA. Pesquisa Operacional Para O Desenvolvimento, 9(2), 48-61. Disponível em: <https://doi.org/10.4322/PODes.2017.005> Acesso em: 3 de dez. de 2020.

[13] FUCHIGAMI, H. (2019). Introdução à linguagem Julia para Programação Matemática. Disponível em: <https://www.fc.unesp.br/Home/Departamentos/Matematica/ermac2019/fuchigami-2019-minicurso-julia-ermac-bauru.pdf> Acesso em: 29 de nov. 2020.

[14] CAIANO, M. M. (2018). Aplicação do Problema do Caixeiro Viajante numa empresa de distribuição: A Heurística de Lin Kernighan. Disponível em: <http://hdl.handle.net/10316/84465> Acesso em 20 de nov. de 2020

[15] TAUFER, F. S. G. (2012) Análise dos operadores de cruzamento do algoritmo genético aplicado ao problema do caixeiro viajante. Disponível em: <http://www.abepro.org.br/biblioteca/enegep2011_tn_sto_140_885_18795.pdf>Acessoem18denov.de2020>

[16] How to Export DataFrame to CSV in Julia. Data to Fish, 2020. Disponível em: <https://datatofish.com/export-dataframe-to-csv-julia/> Acesso em: 4 de dez. 2020.

[17] How to Install a Package in Julia (Example Included). Data to Fish, 2019. Disponível em: <https://datatofish.com/install-package-julia/> Acesso em: 4 de dez. 2020.

[18] ROA, Carlos R. Distância entre locais (Latitude e Longitude). Academicos do excel, 2019. Disponível em: <http://academicosdoexcel.com.br/2017/10/01/distancia-entre-locais-latitude-e-longitude/#:~:text=A%20latitude%20%C3%A9%20a%20dist%C3%A2ncia,)%20e%20ocidental%20(oeste)> Acesso em: 6 de dez. 2020.

[19] Ronis_BR. [ANN] PrettyTables.jl - Print formatted tables in Julia. Julia Lang, 2019. Disponível em: <https://discourse.julialang.org/t/ann-prettytables-jl-print-formatted-tables-in-julia/19550> Acesso em: 1 de dez. 2020.

[20] Tables.jl Documentation. Tables. Disponível em: <https://tables.juliadata.org/stable/> Acesso em: 2 de dez. 2020.

[21] Quick Start Guide. JuMP. Disponível em: <https://jump.dev/JuMP.jl/v0.21.1/quickstart/> Acesso em: 4 de dez. 2020.

[22] GLPK.jl. Julia, 2013. Disponível em: <https://juliapackages.com/p/glpk> Acesso em: 5 de dez. 2020.

[23] BOBCASSELS. What's the best way to optimize findmax with a function?. Julia Lang, 2016. Disponível em: <https://discourse.julialang.org/t/whats-the-best-way-to-optimize-findmax-with-a-function/39606> Acesso em: 8 de dez. 2020.

[24] KAMINSKI, Bogumil. The power of lookup functions in Julia. GitHub, 2020. Disponível em: <https://bkamins.github.io/julialang/2020/10/23/argmax.html> Acesso em: 8 de dez. 2020.

[25] Fórmula de Haversine. Wikipédia, 2020. Disponível em: <https://pt.wikipedia.org/wiki/F%C3%B3rmula_de_Haversine#:~:text=O%20nome%20haversine%20foi%20criado sen2(%CE%B82)> Acesso em: 9 de dez. 2020.

[26] HAESSIG, Pierre. JuMP/GLPK with equal upper and lower bounds. Julia Lang, 2019. Disponível em: <https://discourse.julialang.org/t/jump-glpk-with-equal-upper-and-lower-bounds/28469> Acesso em: 6 de dez. 2020.

[27] ODOW. GLPK.jl. GitHub, 2020. Disponível em: <https://github.com/jump-dev/GLPK.jl> Acesso em: 10 de dez. 2020.

[28] PHANSON, Eric. TravelingSalesmanExact.jl. GitHub, 2020. Disponível em: <https://github.com/ericphanson/TravelingSalesmanExact.jl> Acesso em: 8 de dez. 2020.

[29] Khan Academy Brasil. Provando o Mínimo Quadrado na Regressão Linear (Parte 1). Disponível em: <https://www.youtube.com/watch?v=b8qCqDP7I1I> Acesso em 30 de nov. 2020.

[30] Khan Academy Brasil. Provando o Mínimo Quadrado na Regressão Linear (Parte 2). Disponível em: <https://www.youtube.com/watch?v=yP3ZObjes2E> Acesso em 30 de nov. 2020.

[31] Khan Academy Brasil. Provando o Mínimo Quadrado na Regressão Linear (Parte 3). Disponível em: <https://www.youtube.com/watch?v=Wy6cgZ3LBas> Acesso em 30 de nov. 2020.

[32] Khan Academy Brasil. Provando o Mínimo Quadrado na Regressão Linear (Parte 4). Disponível em: <https://www.youtube.com/watch?v=1uHSxGCDyoU> Acesso em 30 de nov. 2020.