UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

BRUNO HRYNIEWICZ SANTOS CRUZ
FERNANDO DA SILVA FRANÇA

BIBLIOTECA PARA EXEMPLIFICAÇÃO DE ÁLGEBRA LINEAR EASYLINALG

RIO DE JANEIRO
2023

BRUNO HRYNIEWICZ SANTOS CRUZ
FERNANDO DA SILVA FRANÇA

BIBLIOTECA PARA EXEMPLIFICAÇÃO DE ÁLGEBRA LINEAR EASYLINALG

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. João Antonio Recio da Paixão
Co-orientador: Laura de Oliveira F. Moraes

RIO DE JANEIRO
2023

C957b

Cruz, Bruno Hryniewicz Santos
Biblioteca para exemplificação de álgebra linear EasyLinalg / Bruno Hryniewicz Santos Cruz e Fernando da Silva França. – 2023.

93 f.

Orientador: João Antonio Recio da Paixão.
Coorientadora: Laura de Oliveira F. Moraes.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2023.

1. Julia. 2. Álgebra linear. 3. Educação. I. França, Fernando da Silva. II. Paixão, João Antonio Recio da (Orient). III. Moraes, Laura de Oliveira F. (Coorient). IV. Universidade Federal do Rio de Janeiro, Instituto de Computação. V. Título.

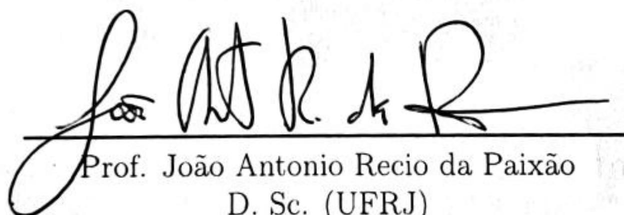
BRUNO HRYNIEWICZ SANTOS CRUZ
FERNANDO DA SILVA FRANÇA

BIBLIOTECA PARA EXEMPLIFICAÇÃO DE ÁLGEBRA LINEAR
EASYLINALG

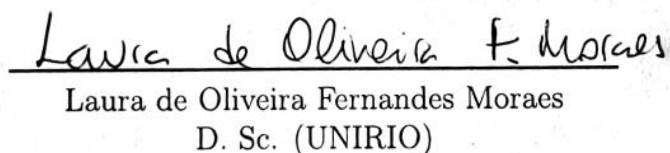
Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 02 de março de 2023

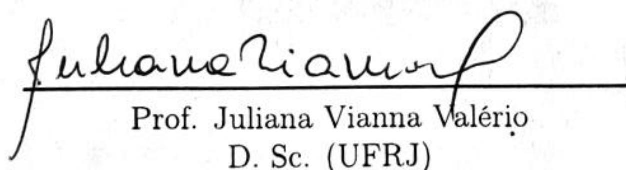
BANCA EXAMINADORA:



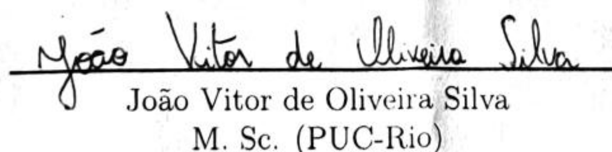
Prof. João Antonio Recio da Paixão
D. Sc. (UFRJ)



Laura de Oliveira Fernandes Moraes
D. Sc. (UNIRIO)



Prof. Juliana Vianna Valério
D. Sc. (UFRJ)



João Vitor de Oliveira Silva
M. Sc. (PUC-Rio)

Dedico aos meus Pais: Jacqueline Hryniewicz dos Santos Cruz e Marcelo Chaves Cruz, que fizeram disso possível, que me incentivaram a fazer o que eu amo e me deram todas as ferramentas para atingir meus objetivos.

Dedico a minha companheira Luiza Nunes Teich, que me acolheu em todos os momentos difíceis e me deu forças para sempre dar o próximo passo. A ela devo o fato de ter descoberto na computação a minha paixão profissional, ter aprendido inglês e todos os outros objetivos que vem me ajudando a conquistar desde 2014.

Dedico as minhas gatas Phoebe e Lia, que foram minhas companheiras durante o desenvolvimento deste trabalho, estando sempre ao meu lado e me fazendo sorrir nos momentos de maiores tensões.

Dedico a minha incrível família, que sempre foi base para minha caminhada.

Em memória dos meus avós que se orgulhariam de onde eu cheguei. Em memória do meu primo Rhafael Hryniewicz Samaha de Carvalho, que estaria junto comigo comemorando esta vitória.

(Bruno Hryniewicz dos Santos Cruz)

Dedico este TCC a meus pais, Roselene da Silva e Antonio França, por todo apoio, aconselhamento, paciência, investimentos de tempo e dedicação, auxílio e amor incondicional, me apoiando inclusive em algumas trocas de curso e universidade, algumas trocas de residência para que eu pudesse alcançar um de meus sonhos.

Dedico a meu irmão mais velho, Felipe França, que serve de inspiração para mim até hoje como pessoa dedicada, empenhada e disciplinada, que foi meu mentor por vários anos tanto de vida pessoal quanto acadêmica, sendo entre nós dois o primeiro a ter de explorar vários caminhos por conta própria e me contar sobre suas experiências para que eu não passasse por águas não navegadas.

Dedico a meus amigos, que souberam puxar minha orelha nos momentos certos, bem como me confortar quando eu precisava e não me abandonaram em momentos críticos, jamais esquecerei os que somaram ao meu lado. De uma coisa tenho convicção, se cheguei até aqui, não foi sozinho.

Em memória a meu avô materno, Raimundo, que deixou toda sua família no estado de nascimento, Ceará, para vir ao Rio de Janeiro quando fez 18 anos, apenas com documento no bolso e um sonho. Ele não teve infância nem adolescência para aproveitar, pois precisava trabalhar, e me dizia que uma caneta era mais leve que uma enxada e eu nunca pude contestar, pois sequer peguei em uma enxada em vida.

A todos que contribuíram positivamente para mim, obrigado por me auxiliarem a realizar meu sonho em me tornar Cientista da Computação.

(Fernando da Silva França)

AGRADECIMENTOS

Agradecemos aos nossos orientadores, João Antonio Recio da Paixão e Laura de Oliveira Fernandes Moraes, por todo suporte durante essa jornada de elaboração do Trabalho de Conclusão de Curso. Agradecemos à Luiza Nunes Teich pelo suporte durante o trabalho, revisando nosso texto e auxiliando na elaboração de algumas ilustrações do trabalho. Obrigado a todos os profissionais e colegas da UFRJ que contribuíram durante toda nossa formação.

"We can only see a short distance ahead,
but we can see plenty there that needs to be done.."

Alan Turing

RESUMO

Matemática e computação são ciências completamente interligadas e que se complementam em diversas linhas de pesquisa. A ciência da computação utiliza do campo matemático da álgebra linear para avançar em diversas de suas ramificações de estudo. Então é de grande importância que o aluno de ciência da computação construa um conhecimento sólido na área de álgebra linear, uma vez que temas como aprendizado de máquina, computação gráfica, entre outros, são estreitamente ligados ao tópico. Porém, demonstrar a importância dessa área para os estudantes e os motivar neste tema não é uma tarefa trivial, podendo se tornar um desafio ao professor. Com este trabalho, construímos uma biblioteca de Julia Lang, chamada EasyLinalg, para facilitar a exemplificação de temas da álgebra linear. Tratando a exemplificação como uma maneira de motivar e melhorar resultados no estudo, este projeto auxiliará no início dos estudos de álgebra linear. Esta construção objetiva melhorar a dinamicidade dos exemplos de aula, possibilitando o professor a construir exemplos de maneira simples e utilizando objetos de estudo mais próximos a aplicações reais, como imagens e sinais. A biblioteca também permite ao aluno, no que lhe concerne, ter autonomia de criar e modificar exemplos. EasyLinalg cria então uma maneira de introduzir os benefícios de se utilizar programação para fazer exemplos de álgebra linear, não adicionando a complexidade de efetivamente programar tais exemplos. EasyLinalg traz a possibilidade de se construir exemplos com pontos no $\mathbb{R}^2$ e $\mathbb{R}^3$, vetores $\mathbb{R}^2$ e $\mathbb{R}^3$, sinais e imagens. Sendo a biblioteca de código aberto, este trabalho introduz a primeira versão de sua implementação e deixa aberto ao público a possibilidade da evolução da mesma.

Palavras-chave: Julia; álgebra linear; exemplos; educação

ABSTRACT

Mathematics and computer science are completely interconnected sciences that complete each other in different research perspectives. Computer science uses mathematics regarding linear algebra in order to advance in some lines of research. It is of great importance that a computer science student builds solid knowledge in linear algebra, since subjects such as machine learning, computer graphics, among others, are directly linked to the topic. However, highlighting the importance of learning this particular topic may be challenging to professors. Through the present work, we have developed a library called EasyLinalg, written in Julia Lang, to favor the demonstration of linear algebra exercises. Treating this simplification in demonstration as a way to increase motivation and enhance results while studying, this project aims to help students in the early stages of linear algebra. The objective construction in the library helps the dynamism of classes exercises, assisting professors when creating exercises, also using study subjects closer to reality, such as images and signals. This library also assists students to explore and have autonomy after studying some topics in classes, since they are able to create and modify exercises themselves. EasyLinalg creates a possibility when introducing the benefits of studying using programming language, while does not necessarily make the users code such exercises from scratch. EasyLinalg makes possible to create examples using points and vectors in $\mathbb{R}^2$ and $\mathbb{R}^3$, signals am images. Since the library is open source, this project presents its first public version and leaves the public the possibility to keep on developing as needed.

Keywords: Julia; linear algebra; examples; education

LISTA DE ILUSTRAÇÕES

Figura 1 – Ponto (1,2) . . . 19
Figura 2 – Pontos (1,2) e (1,-2) . . . 20
Figura 3 – Imagem Eniac . . . 21
Figura 4 – Imagem Eniac refletida . . . 22
Figura 5 – Diagrama de iteração planejado . . . 24
Figura 6 – Proto-persona do Professor . . . 27
Figura 7 – Proto-persona do Aluno . . . 28
Figura 8 – Proto-persona da Programadora . . . 29
Figura 9 – Draw para seta 2D . . . 33
Figura 10 – Draw para vetor de setas 2D . . . 33
Figura 11 – Draw de seta 3D . . . 34
Figura 12 – Draw para vetor de setas 3D . . . 34
Figura 13 – Draw de ponto 2D . . . 35
Figura 14 – Draw para vetor de pontos 2D . . . 35
Figura 15 – Draw de ponto 3D . . . 36
Figura 16 – Draw para vetor de pontos 3D . . . 36
Figura 17 – Draw de um sinal . . . 37
Figura 18 – Draw para vetor de sinais . . . 37
Figura 19 – Draw de imagem . . . 38
Figura 20 – Draw para vetor de imagens . . . 38
Figura 21 – Matriz A . . . 39
Figura 22 – Ponto B . . . 39
Figura 23 – Ponto C . . . 40
Figura 24 – Ponto B e Ponto C . . . 40
Figura 25 – Imagem e Código . . . 41
Figura 26 – Imagem e Código . . . 42
Figura 27 – Exemplo de média aritmética de Pontos 2D . . . 43
Figura 28 – Exemplo de média aritmética de Pontos 3D . . . 44
Figura 29 – Exemplo de média aritmética de Setas 2D . . . 44
Figura 30 – Exemplo de média aritmética de Setas 3D . . . 45
Figura 31 – Exemplo de média aritmética de Sinais . . . 45
Figura 32 – Exemplo de média aritmética de Imagens . . . 46
Figura 33 – Exemplo de morphing para Pontos 2D . . . 47
Figura 34 – Exemplo de morphing para Pontos 3D . . . 48
Figura 35 – Exemplo de morphing para Setas 2D . . . 48
Figura 36 – Exemplo de morphing para Setas 3D . . . 49

Figura 37 – Imagens e Sinais para exemplo de morphing 49
Figura 38 – Morphing de Imagens e Sinais 50
Figura 39 – Exemplo de transformação linear de reflexão para Pontos 2D 51
Figura 40 – Exemplo de transformação linear de reflexão para Pontos 3D 51
Figura 41 – Exemplo de transformação linear de reflexão para Setas 2D 52
Figura 42 – Exemplo de transformação linear de reflexão para Setas 3D 52
Figura 43 – Exemplo de transformação linear de reflexão para Imagens 53
Figura 44 – Transformação linear de cisalhamento 54
Figura 45 – Exemplo A X B = C 55
Figura 46 – Componentes A e C 55
Figura 47 – Componente B, resultado para exemplo de decomposição de Pontos 2D 56
Figura 48 – Componentes A e C 56
Figura 49 – Componente B, resultado para exemplo de decomposição de Pontos 3D 57
Figura 50 – Componentes A e C 57
Figura 51 – Componente B, resultado para exemplo de decomposição de Setas 2D . 58
Figura 52 – Componentes A e C 58
Figura 53 – Componente B, resultado para exemplo de decomposição de Setas 3D . 59
Figura 54 – Componentes A e C 59
Figura 55 – Componente B, resultado para exemplo de decomposição de Sinais . . 60
Figura 56 – Cadeia de Markov em Pontos 2D 61
Figura 57 – Cadeia de Markov em Pontos 3D 62
Figura 58 – Cadeia de Markov em Setas 2D 63
Figura 59 – Cadeia de Markov em Setas 3D 64
Figura 60 – Sinal inicial para exemplo de Cadeia de Markov 65
Figura 61 – Aplicação da Cadeia de Markov em sinal 66
Figura 62 – Sinal após convergência 67
Figura 63 – Resultado dos testes unitários 89

LISTA DE CÓDIGOS

Código 1 Zero Lógico 70
Código 2 Estrutura ponto 2D 77
Código 3 Redefinições ponto 2D 78
Código 4 Estrutura ponto 3D 78
Código 5 Estrutura seta 2D 79
Código 6 Estrutura seta 3D 79
Código 7 Estrutura sinal 80
Código 8 SignalBuildU 80
Código 9 SignalBuild 80
Código 10 Redefinindo sinal 81
Código 11 Estrutura imagem 82
Código 12 Redefinindo imagem 83
Código 13 draws.jl 85
Código 14 operations-example.jl 86
Código 15 product.jl 87
Código 16 arrow2DTeste.jl 88

LISTA DE TABELAS

Tabela 1 – Tipos implementados na EasyLinalg 31
Tabela 2 – Operações implementadas na EasyLinalg 32

LISTA DE ABREVIATURAS E SIGLAS

E/S	Entrada e Saída
IA	Inteligência artificial
API	Application Programming Interface
RGB	Red, Green and Blue
GPS	Sistema de Posicionamento Global

SUMÁRIO

1 INTRODUÇÃO . . . 15
1.1 ORGANIZAÇÃO . . . **15**

2 MOTIVAÇÃO . . . 17
2.1 IMPORTÂNCIA DA ÁLGEBRA LINEAR . . . **17**
2.2 FALTA DE MOTIVAÇÃO . . . **17**
2.3 COMO QUEREMOS AJUDAR A RESOLVER A FALTA DE MOTIVAÇÃO . . . **18**
2.3.1 Exemplo algébrico de reflexão de um ponto . . . 19
2.3.2 Exemplo geométrico de reflexão de um ponto . . . 19
2.3.3 Exemplo prático de utilização de reflexão . . . 20
2.4 VIABILIDADE DE MAIS EXEMPLOS . . . **22**

3 EASYLINALG PARA OS USUÁRIOS . . . 23
3.1 PROPOSTA . . . **23**
3.1.1 Entrada e Saída . . . 24
3.1.2 Abstração . . . 26
3.2 USUÁRIOS DA EASYLINALG . . . **26**
3.2.1 Proto-persona do Professor . . . 27
3.2.2 Proto-persona do Aluno . . . 27
3.2.3 Proto-persona da Programadora . . . 28
3.3 MANUAL DE UTILIZAÇÃO . . . **29**
3.3.1 Instalação . . . 29
3.3.2 Entrada de dados e Tipos de dados . . . 30
3.3.3 Operações . . . 31
3.3.4 Saída de dados . . . 32
3.3.5 Aula Exemplo . . . 38
3.4 RESULTADOS . . . **42**
3.4.1 Exemplo de multiplicação por escalar e soma: média aritmética 43
3.4.2 Exemplo combinação convexa: morphing . . . 47
3.4.3 Exemplo de multiplicação por Matriz: transformações lineares 50
3.4.4 Exemplo de resolução de sistemas lineares . . . 54
3.4.5 Exemplo de cadeias de Markov . . . 60

4 EASYLINALG PARA DESENVOLVEDORES . . . 68
4.1 MOTIVO DA ESCOLHA DA JULIA LANG . . . **68**

4.2 DESIGN TÉCNICO 68
4.2.1 Motivadores 69
4.2.2 Código Aberto 69
4.2.3 Tipos de Dados 70
4.2.4 Heranças 71
4.2.5 Polimorfismo 72
4.2.6 Árvore de arquivos 73
4.2.7 Testes unitários 76
4.3 IMPLEMENTAÇÃO 77
4.3.1 Tipos definidos 77
4.3.2 Draws 84
4.3.3 Operações internas 85
4.3.4 Testes Unitários 87

5 CONSIDERAÇÕES FINAIS 90
5.1 CONCLUSÃO 90
5.2 EVOLUÇÃO E TRABALHOS FUTUROS 91

REFERÊNCIAS 92

1 INTRODUÇÃO

Existe uma grande interseção entre matemática e computação, dado que o nascimento de computadores foi motivado pela necessidade de resolver problemas matemáticos complexos. Com o avanço da ciência da computação, tais interseções acabam sendo ocultadas por camadas sobrepostas de abstração, camuflando o conceito matemático de resolver problemas contidos no ato de criar *softwares* (CAMPBELL-KELLY, 2009). Esses níveis de abstrações são essenciais para o avanço tecnológico, visto que facilitam a interação homem-máquina. Um exemplo disso é como a adição de uma interface gráfica em sistemas computacionais conseguiu facilitar sua utilização apenas criando uma abstração em cima do que antes era apenas linha de comando. Com este aparente distanciamento entre a computação contemporânea e sua origem matemática, não fica claro para os novos estudantes da área a necessidade de tantas cadeiras de matemática em seu curso (BEAUBOUEF, 2002). Sob este pretexto, existem alunos que não se motivam a estudar tais temas durante suas graduações e podem com isso ter seus desempenhos acadêmico e profissional prejudicados (BEAUBOUEF, 2002).

Torna-se um desafio ao corpo acadêmico diminuir o distanciamento entre matemática e computação. Uma abordagem que pode ser utilizada para isso é a exemplificação (AYTEKIN; KIYMAZ, 2019). Tendo em mente essa provocação, a biblioteca EasyLinalg é pensada como uma ferramenta para auxílio no ensino de Álgebra Linear, buscando aplicar uma nova camada de abstração de modo a facilitar uma exemplificação mais próxima à utilização em problemas reais.

1.1 ORGANIZAÇÃO

A seguir será especificado a organização do texto deste trabalho, a fim de facilitar a navegação pelo conteúdo deste trabalho.

- O Capítulo 2 levanta os pontos que motivaram o trabalho e a abordagem que este trabalho visa tomar com o propósito de facilitar demonstrações dos conceitos teóricos.

- O Capítulo 3 aborda a proposta concreta do projeto e conceitos-chave da implementação com foco no usuário alvo: alunos e professores de álgebra linear. Por fim, este capítulo apresenta um manual de utilização da biblioteca para os usuários. Neste ainda são demonstrados os resultados da biblioteca.

- O Capítulo 4 tem a proposta de destrinchar as decisões técnicas tomadas no projeto e ainda como foram efetivamente implementadas. Seu ponto focal é em um público alvo mais técnico que deseja contribuir para o projeto.

- Finalizando este trabalho, o Capítulo 5 sintetiza o que foi abordado neste trabalho e apresenta a visão dos desenvolvedores para os trabalhos futuros.

2 MOTIVAÇÃO

2.1 IMPORTÂNCIA DA ÁLGEBRA LINEAR

Vários fenômenos do nosso mundo podem ser modelados por equações e funções lineares. O poder efetivo dessa ramificação da matemática é incalculável, visto que modelagem linear é comum em múltiplas linhas de pesquisa acadêmica e de mercado (STRANG, 2006). Entender o computador como uma plataforma para potencializar o uso da álgebra linear é uma interpretação válida, mas também é importante ressaltar que a própria computação utiliza da álgebra linear como uma ferramenta para a evolução de sua ciência.

Dado então a vastidão desta área matemática, esta sessão terá como escopo introduzir aplicações da álgebra linear sobre o foco da ciência da computação e, através disso, mostrar a importância da álgebra linear para o estudante de computação. Para tal objetivo, serão abordados três de utilizações contemporâneas de álgebra linear em aplicações de computação.

- Aplicativos de “GPS” que fornecem cálculo de melhor rota, onde se utiliza da resolução de problemas de caminhos mínimos e otimização (CALAFIORE; GHAOUI, 2014) para recomendar o trajeto mais rápido.
- A computação gráfica, é uma área de pesquisa que tem como fundamento a álgebra linear. Alguns produtos construídos com base na computação gráfica são jogos eletrônicos, filmes, simuladores, imagens médicas e visualizações científicas (SHIRLEY; ASHIKHMIN; MARSCHNER, 2009).
- Aplicações de aprendizado de máquina utilizam álgebra linear como ferramenta de sua construção (AGGARWAL; AGGARWAL; LAGERSTROM-FIFE, 2020), então aplicações como processamento de linguagem natural, detecção de fraudes e tantos outros tópicos têm suas bases na álgebra linear.

Assim como nos exemplos acima, diversas outras linhas de pesquisa na computação utilizam de álgebra linear, sendo então esta uma evidência da importância do tópico na formação de cientistas da computação.

2.2 FALTA DE MOTIVAÇÃO

Atualmente, grandes quantias financeiras estão sendo constantemente injetadas em empresas de base tecnológica (CORNELLI et al., 2020), assim novas empresas de tecnologia surgem diariamente e necessitam de mão de obra. Então o mercado atual se torna um grande atrativo para se estudar computação, devido à abundância de vagas e salários

muitas vezes acima do salário mínimo (JORGE; COSTA, 2021). Historicamente, os jovens do Brasil apresentam déficits na competência matemática quando comparado à média da Organização para a Cooperação e Desenvolvimento Econômico (LIMA et al., 2020), sendo este um dos motivadores que cria nos professores a percepção da dificuldade no ensino de matemática para o ensino superior (MASOLA; ALLEVATO, 2016). Aliado a este déficit, há uma falsa impressão da não necessidade da matemática na formação da computação. Alunos entram no curso atraídos pelas ofertas do mercado de trabalho e mantêm seu foco em entender quase exclusivamente programação. Assim, acabam realizando as cadeiras matemáticas apenas por obrigação, pois não percebem ou não dão a devida importância às suas aplicações na computação. Estas cadeiras matemáticas se tornam grande parte da dificuldade do curso, fato este, evidenciado pelos grandes índices de reprovação das cadeiras de cálculo diferencial e integral da UFRJ (LOPEZ; SEGADAS, 2014). Esses pontos influenciam diretamente a motivação dos alunos.

Aprender algo novo pode ser muito estimulante, principalmente se a vontade de aprender partir do próprio individuo. Entretanto, ficar para trás em relação à turma, sentir que não está conseguindo absorver o conteúdo e ter dificuldades para resolver os exercícios mais básicos pode causar uma falta de motivação no processo de aprender. O ensino da álgebra linear, em particular no curso de Ciência da Computação da Universidade Federal do Rio de Janeiro, acontece no terceiro período letivo e neste ponto do curso, o aluno já teve contato com cadeiras tanto de matemática quanto de programação. Dado a grande taxa de reprovação que estes cursos carregam (LOPEZ; SEGADAS, 2014), há grandes chances do aluno já ter passado por dificuldades nos cursos de cálculo. Seu desempenho anterior pode afetar diretamente a competência percebida que estes alunos têm sobre suas capacidades matemáticas, assim influenciando diretamente em seu desempenho futuro (SIMÕES; FERRÃO, 2005). Ao iniciar o curso de álgebra linear, o aluno pode enxergar a cadeira como mais uma em que terá dificuldades e que apenas precisa cumprir por formalidade, visto que sua utilização na computação ainda não é clara para um aluno de terceiro período.

2.3 COMO QUEREMOS AJUDAR A RESOLVER A FALTA DE MOTIVAÇÃO

Quebrar a barreira desta pseudo-separação entre programação e matemática pode ser uma maneira de criar motivação nos alunos ainda em início de curso. Para isso, apresentar mais aplicações de álgebra linear na computação, com exemplos próximos a aplicações reais, pode ser um fator que aumenta a motivação dos mesmos. Assim, como proposta, a EasyLinalg buscará uma exemplificação próxima à demonstração a seguir:

2.3.1 Exemplo algébrico de reflexão de um ponto

Ao explicar uma transformação linear de reflexão, por exemplo, podemos dizer que a matriz:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

espelha um ponto do $\mathbb{R}^2$ ao redor do eixo X, ou seja, se multiplicarmos um ponto do $\mathbb{R}^2$ por essa matriz, ele terá sua coordenada Y invertida:

$$\begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \end{bmatrix}$$

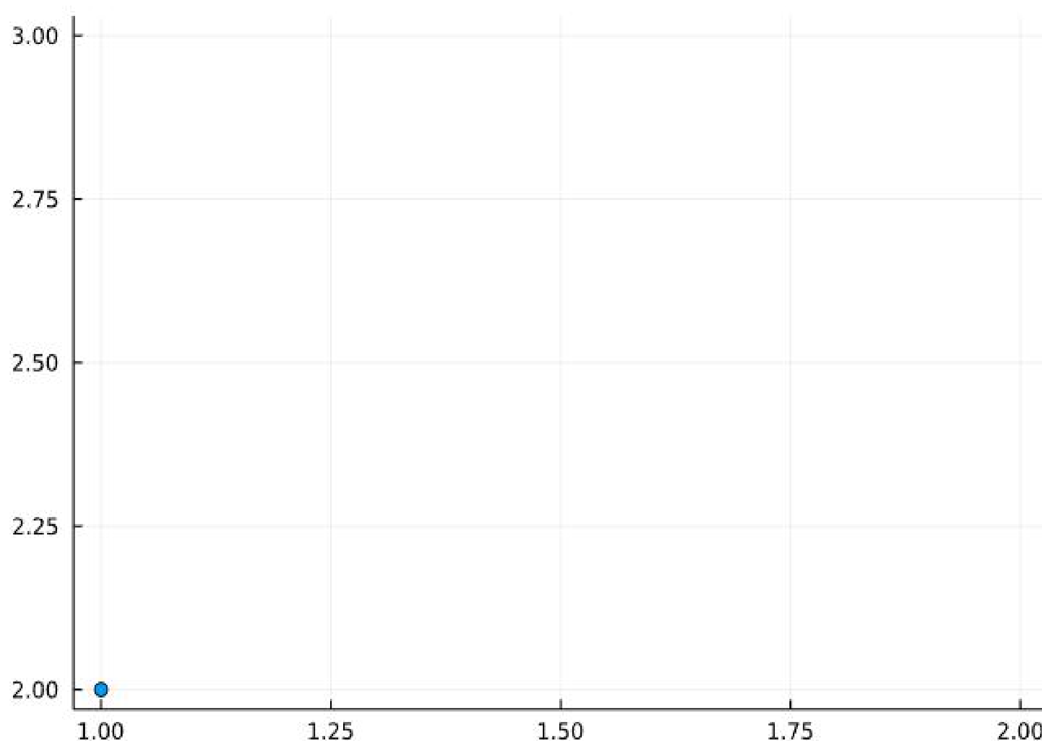
2.3.2 Exemplo geométrico de reflexão de um ponto

Após o exemplo numérico da reflexão, podemos enriquecê-lo com um gráfico 2D do que aconteceu. Suponha esse mesmo ponto em um gráfico no $\mathbb{R}^2$. O ponto representado de forma algébrica é:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Sua representação geométrica no plano é:

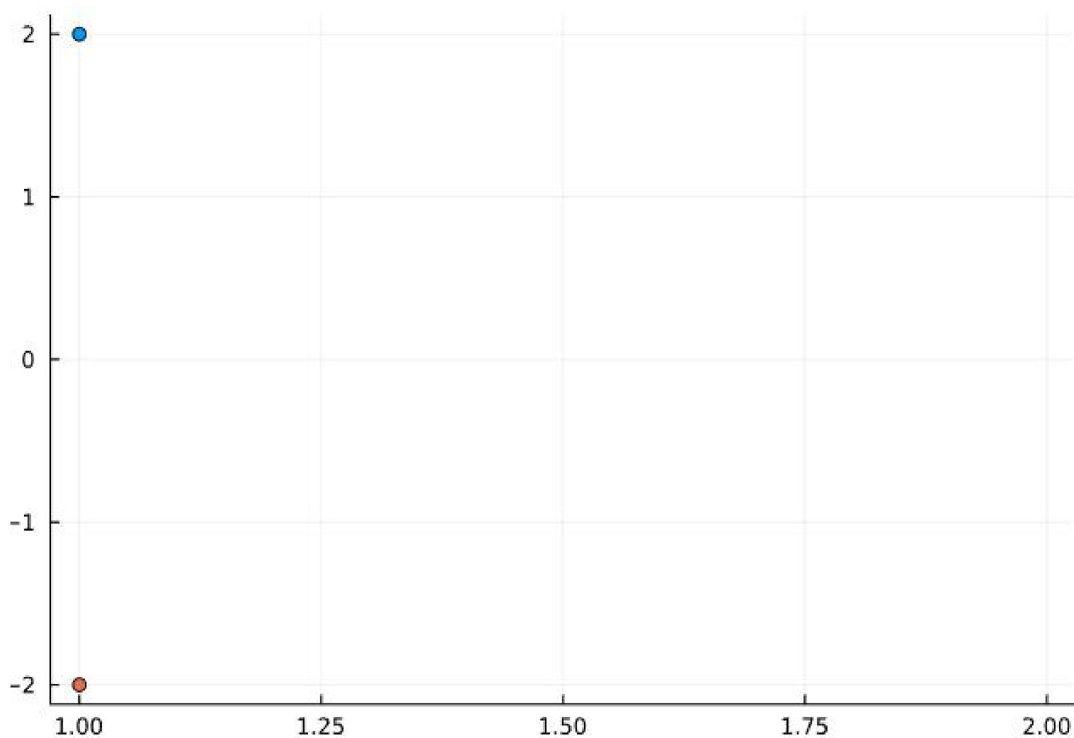
Figura 1 – Ponto (1,2)



Ponto (1,2) no R^2

Aplicando o espelhamento desse ponto ao redor do eixo X seria:

Figura 2 – Pontos (1,2) e (1,-2)

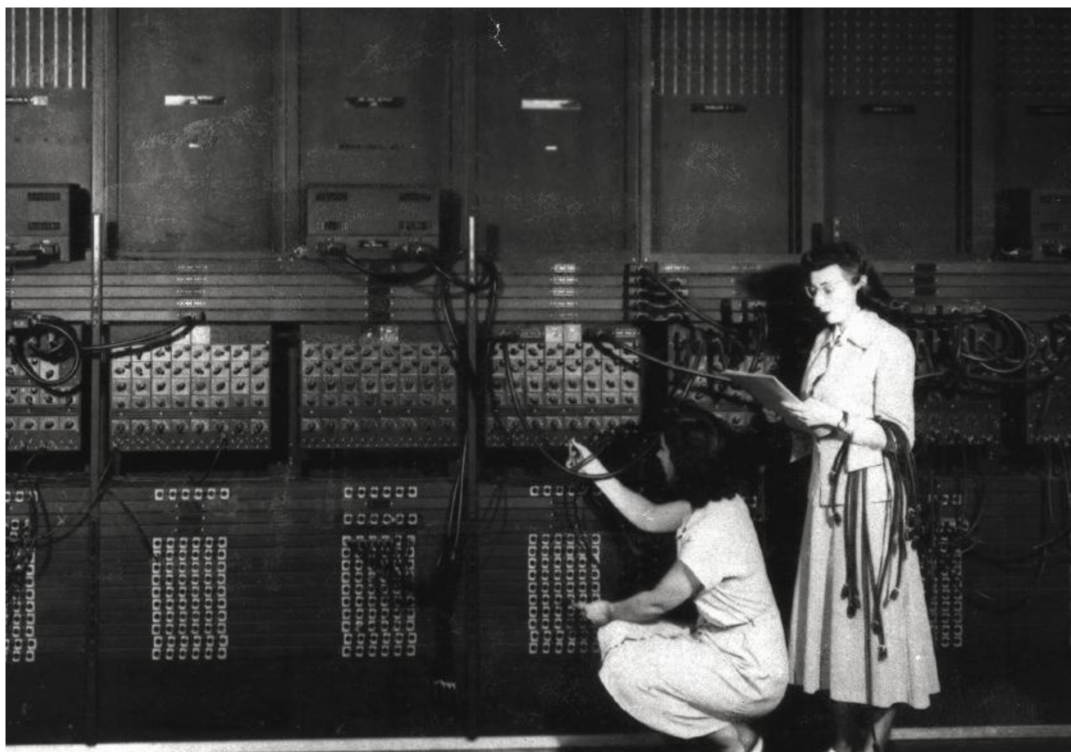


Pontos (1,2) e (1,-2) no R^2

2.3.3 Exemplo prático de utilização de reflexão

Por fim, entendida a parte algébrica e geométrica, podemos então partir para uma exemplificação de utilização deste conceito. Para isso utilizaremos a imagem:

Figura 3 – Imagem Eniac



Ester Gerston e Gloria Gordon, programadoras do ENIAC (créditos: ARL Technical Library / U.S. Army)

Uma imagem é computacionalmente representada por uma matriz de pixels, sendo um píxel um ponto no $\mathbb{R}^2$ com uma intensidade de cor associada a ele. Então pode-se aplicar em cada píxel da imagem a transformação vista anteriormente e o resultado será a reflexão da imagem, considerando o centro da foto como origem de um plano cartesiano. Entretanto, existe uma particularidade quando se trata de imagens. Quando falamos geometricamente do ponto [1, 2], é entendido que a coordenada X deste ponto vale 1 e a coordenada Y vale 2. Já no âmbito de computação gráfica, a representação dos eixos X e Y do plano cartesiano convencional é invertida. Quando falamos do píxel de posição [1, 2], estamos falando de um píxel de coordenada X com valor 2 e coordenada Y de valor 1. Esse é o padrão da representação de imagens em computadores, e dado este padrão, ao aplicar a mesma matriz anterior nos píxeis da imagem, teremos uma reflexão ao redor do eixo Y e não no eixo X.

Seguindo com o exemplo, aplicaremos em cada píxel da imagem a matriz de reflexão em torno do eixo Y:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Obtendo então o seguinte resultado:

Figura 4 – Imagem Eniac refletida

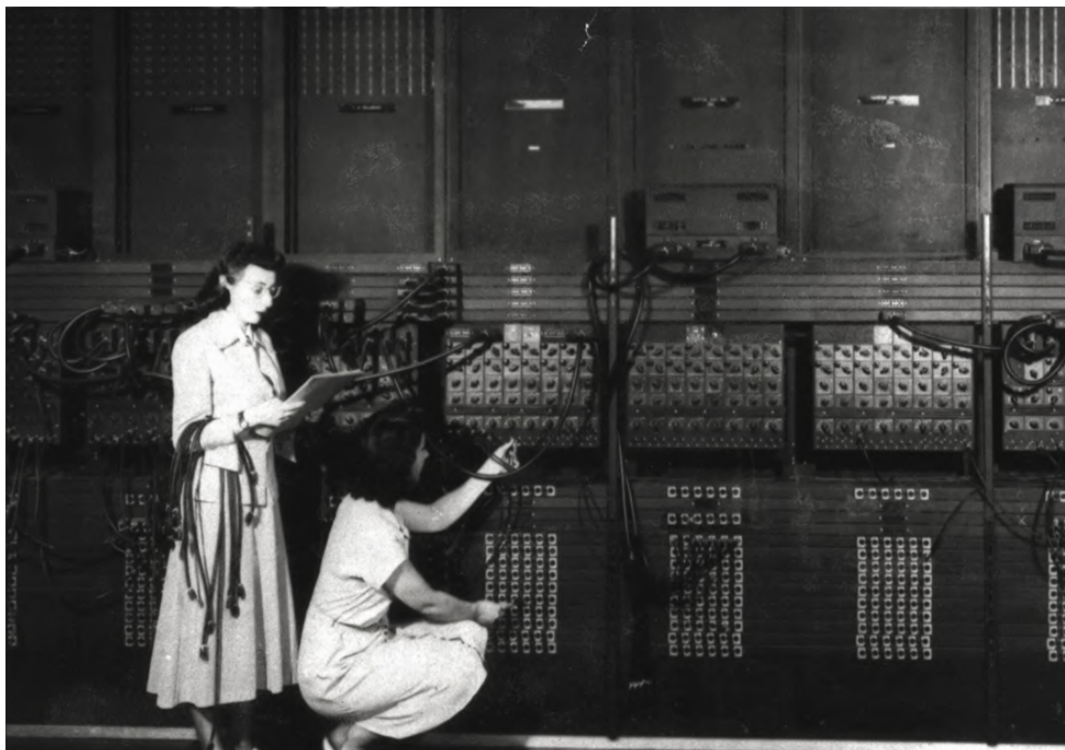


Imagem refletida no eixo Y

2.4 VIABILIDADE DE MAIS EXEMPLOS

Visando agir diretamente na falta de motivação, mostrar melhores exemplos para alunos é um caminho efetivo a se seguir (AYTEKIN; KIYMAZ, 2019). Porém, a possibilidade de mostrar mais aplicações é normalmente limitada pela complexidade técnica por trás de sua implementação. Utilizar exemplos ligados à computação exige um nível de conhecimento técnico que por muitas vezes tanto aluno quanto professor podem não ter, afinal, é comum o vir de uma formação anterior em matemática e os alunos estarem no início de seus estudos na área da computação. A compreensão técnica dos exemplos tem que existir em ambas as partes para ser suficiente, visto que uma vez que o professor programe um exemplo, este tem que ser totalmente compreendido pelos alunos.

3 EASYLINALG PARA OS USUÁRIOS

EasyLinalg é uma biblioteca cujo objetivo é facilitar a demonstração de exemplos de álgebra linear, buscando aproximar a teoria da prática em uma tentativa de melhorar a absorção de conhecimento dos estudantes. Atua criando uma camada de abstração, reduzindo os conhecimentos necessários de programação para realizar exemplificações de conteúdos da matéria.

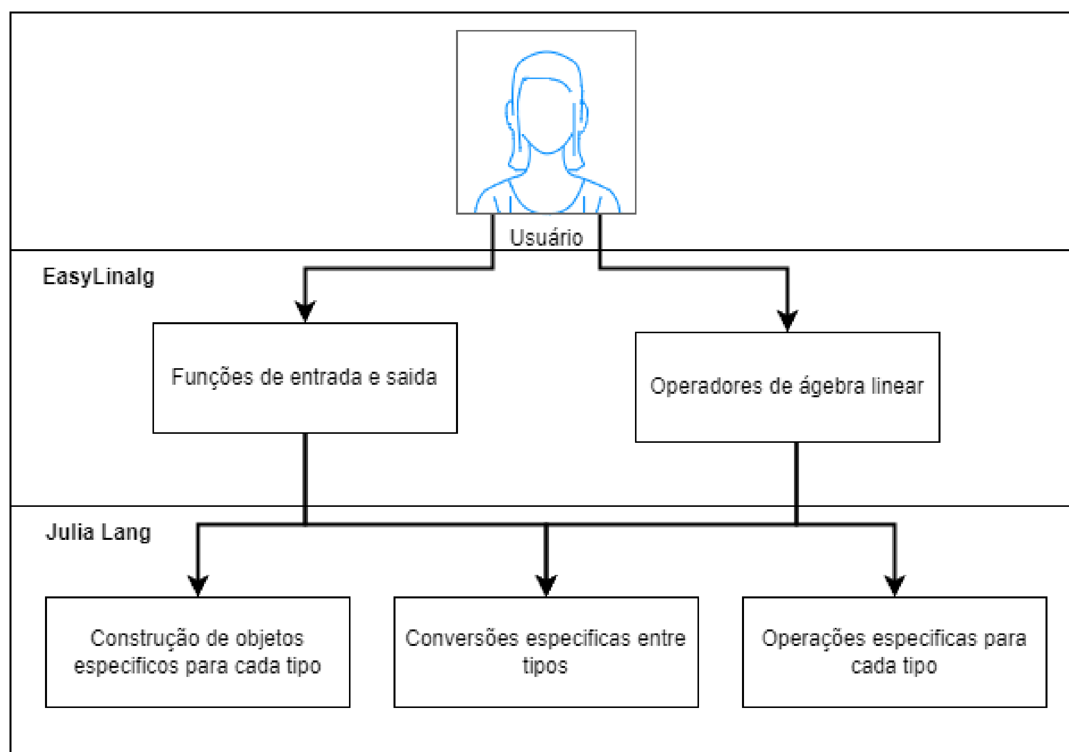
A biblioteca deve então facilitar demonstrações de aplicações da álgebra linear utilizando tipos de dados matriciais que tenham utilizações conhecida para os alunos, como imagens, sinais e vetores, sendo abstraído todo o trabalho de carregar e imprimir essas categorias de dados no computador. Possibilita assim uma pessoa com pouco conhecimento de programação a explorar a essas estruturas da álgebra linear de maneira visual e próxima a aplicações reais.

Outro objetivo da EasyLinalg é promover a autonomia do aluno para testar e experimentar novas possibilidades independentemente dos exemplos dados em sala de aula, ficando a cargo de sua imaginação usar novos parâmetros e valores nos exercícios, por exemplo.

3.1 PROPOSTA

EasyLinalg é uma biblioteca que propõem uma interface de interação entre o usuário e a linguagem de programação Julia. Tal interface é uma estrutura de código composta por tipos de dados, operações de álgebra linear e funções de entrada e saída de dados. Utilizando conceitos de orientação a objetos para criação de tipos de dados com métodos e atributos associados, EasyLinalg carrega nestes tipos suas regras de operação. A Figura 5 ilustra a interação entre usuário e a interface construída no projeto, onde o usuário só precisa utilizar a biblioteca para fazer entrada e saída de dados e operações de álgebra linear em cima destes dados, não tendo visão de todas as operações e implementações que houveram para que as operações pudessem ser feitas da maneira que são. Em uma camada abaixo, a biblioteca interage com as construções da linguagem Julia para possibilitar seu funcionamento, recebendo apenas o que o usuário anteriormente entrou, operações e entrada e saída de dados.

Figura 5 – Diagrama de iteração planejado



Interação do Usuário com a Easylinalg.

Esse projeto foca nos primeiros contatos de alunos com álgebra linear, apresentando uma biblioteca interativa onde alunos e professores podem desfrutar de recursos como operação de pontos, vetores e imagens, sem muitas das complexidades que esses tipos de dados podem apresentar em sua implementação. Faz parte da proposta do projeto descomplicar esse primeiro contato, tentando fazer o aluno ocupar a maior parte do tempo de estudo com foco nos conceitos apresentados no curso de álgebra linear, facilitando a utilização da programação, que poderia ser um obstaculo para seu foco.

Por fim, EasyLinalg não objetiva ser uma biblioteca avançada de álgebra linear e sim implementar um pequeno conjunto de funcionalidades. Seu foco é auxiliar no ensino da matéria para aqueles que estão no início do seu aprendizado no assunto.

3.1.1 Entrada e Saída

Entrada/saída é uma expressão bem conhecida na computação para realizar uma referência à comunicação entre o mundo real e sistemas de computação. A entrada é tudo o que pode ser capturado do mundo externo por periféricos como, por exemplo, a foto capturada por uma câmera, o texto digitado em um teclado, o movimento e o apertar de botões do mouse. Em outras palavras, entrada é o que vem do mundo externo e de alguma maneira é digitalizado e entra no sistema de computação. Já a saída é tudo o que sai do dito sistema para o mundo externo, como o que aparece no monitor, o que é

impresso numa impressora ou saídas de áudio em alto-falantes.

Para contextualizar E/S (sigla comumente utilizada para entrada e saída) na computação, consideremos que desde o princípio das máquinas computacionais E/S existe. Seja no ábaco onde a entrada era o posicionamento das pedras e a saída a interpretação delas, passando pelos famosos computadores programados com cartão perfurado, E/S sempre foi um pilar das máquinas programáveis. Com a evolução da tecnologia, a interação ficou mais simples e intuitiva, além da utilização de dispositivos computacionais se tornar cada vez mais ampla. Basta comparar computadores programados via cartão perfurado, onde somente especialistas conseguiam utilizá-los, com os *smartphones*, com os quais crianças ainda não alfabetizadas conseguem interagir. Análogo ao exemplo anterior, quando falamos de computação cientifica, temos bibliotecas de álgebra linear muito eficientes e com inúmeras ferramentas e possibilidades a explorar, que sendo usadas por especialistas, conseguem gerar excelentes resultados. Porém, quando uma pessoa sem experiência nessas ferramentas tenta se utilizar delas, há uma grande curva de aprendizado para se chegar ao ponto de utilização desejado para o ensino e aprendizagem, se tornando então um tema de estudo paralelo e não um facilitador de aprendizado.

Durante o estudo da álgebra linear, em seu processo de exemplificação, são geralmente utilizados objetos como pontos no espaço, vetores e matrizes. Porém, computacionalmente existem muitos outros objetos com características matriciais que podem ser muito mais ilustrativos para o ensino. A maneira de realizar a entrada e depois exibir esses objetos é uma barreira à utilização de exemplos mais palatáveis, porque afinal, tais objetos serão carregados em vetores genéricos e operados sem conservar nada da sua semântica ou o que eles representam e como devem ser interpretados, deixando então essas interpretações a cargo de quem está programando.

Se pudermos então encapsular estes objetos de maneira a manter sua semântica e característica, e dentro deste encapsulamento já estiver pré-estabelecido como fazer a entrada de dados, como realizar operações com esses dados e como mostrar essas informações, será muito reduzido o nível de complexidade necessária para realizar a entrada e saída de um dado, além de agregar informações semânticas para facilitar a interpretação dos resultados. Essa redução de complexidade pode então ser extrapolada para se trabalhar com objetos matriciais mais complexos, como imagens ou sinais, sem aumentar a complexidade para quem utiliza.

Entrada e saída é um dos principais conceitos explorados pela EasyLinalg, visto que essa é a primeira barreira que um usuário se depara ao tentar utilizar uma linguagem de programação para praticar álgebra linear. Este projeto também visa amenizar esta dificuldade definindo métodos fáceis para entrada e saída como uma de suas premissas.

3.1.2 Abstração

Falando sobre abstração em um âmbito computacional, um computador é uma máquina que interpreta somente a existência ou ausência de energia elétrica em um circuito, ou seja, um sinal eletrônico. Então é efetuado uma abstração em cima desta existência ou ausência de energia, utilizando de zeros e uns, para os interpretar como o conjunto numérico dos binários. Novas camadas de abstração vão sendo empilhadas em cima desta informação binária, assim agregando mais sentidos a ela, a ponto de conseguir então gerar toda interação moderna que temos com computadores sem ao menos precisarmos entender como realizar operações matemáticas com números binários.

As abstrações atuam como uma troca entre liberdade e facilidade, quanto maior o nível de abstração, menor a liberdade e maior a facilidade de uso. Para enxergar esta premissa, basta analisar que o código escrito em binário pode ser utilizado para fazer qualquer tipo de interação com o computador, mas sua dificuldade de escrita faz com que poucos programadores escrevam códigos diretamente desta forma. Em outra ponta, pode-se analisar as interfaces gráficas que, em geral, são intuitivas e práticas, mas somente conseguem fazer aquilo para o que foram programadas, tendo então um nível de liberdade muito menor do que um código binário, por exemplo.

Ferramentas computacionais costumam aplicar esse conceito, como, por exemplo, a linguagem de programação de alto nível Julia, que oferece ao desenvolvedor uma interface de programação mais amigável do que a linguagem de máquina. A fim de facilitar o ensino, a utilização desta abordagem de abstração também é uma estratégia que vem sendo utilizada por ferramentas como o *Scikit-Learn* (VAROQUAUX et al., 2015), que traz um conjunto de componentes de aprendizado de máquina em Python onde algoritmos de alto grau de complexidade estão abstraídos por trás de chamadas de funções.

EasyLinalg se apoia neste pilar, onde é isolado um pequeno conjunto da álgebra linear e em cima dele é criada uma camada que facilita sua utilização. Essa metodologia traz consigo uma troca onde se perde níveis de liberdade de utilização em troca de facilidade nesse uso, sendo completamente aceitável dado o objetivo da biblioteca. A vantagem da abstração no âmbito do ensino é levar o foco para o que está sendo ensinado. Se o objetivo é aprender álgebra linear, programação não deveria ser um obstáculo, então quanto mais abstraída a programação for, mantendo suas vantagens, mais fácil será aprender.

3.2 USUÁRIOS DA EASYLINALG

Ao se projetar a EasyLinalg, entender o público alvo foi um ponto crucial para criar uma biblioteca que tenha potencial de utilização. Para isso, foram definidas três proto-personas (GOTHELF, 2012) de utilização do projeto, sendo elas o Professor, o Aluno e a Desenvolvedora.

3.2.1 Proto-persona do Professor

A proto-persona do Professor, representada na Figura 6, é caracterizada como aquele que está ensinando álgebra linear. Seu conhecimento de programação é, ao menos, básico e seu conhecimento de álgebra linear é avançado. Seu caso de uso na EasyLinalg é desenvolver exemplos de maneira rápida, onde facilmente ele possa alterar as características de seu exemplo, assim trazendo dinamicidade para aula. A EasyLinalg para o professor deve ser um otimizador de tarefas e um simplificador de exemplos, assim tornando mais palpáveis e atraentes os exemplos dados pelo mesmo. Tudo isso ainda traz um ganho de tempo de aula para o professor, que facilmente consegue gerar seus exemplos sem a necessidade de desenhar tanto no quadro.

Figura 6 – Proto-persona do Professor

Infográfico descritivo da proto-persona do professor

3.2.2 Proto-persona do Aluno

O Aluno, representado na Figura 7, é aquele que está aprendendo. Seu conhecimento de álgebra linear ainda é pequeno, seu conhecimento de programação é básico e nunca teve contato com a linguagem Julia. Para o Aluno, seu maior ganho é a simplicidade que a EasyLinalg trará, fazendo com que os exemplos dados pelo professor sejam mais simples de serem entendidos. Essa simplicidade também pode ser convertida em autonomia, assim facilitando o aluno a explorar sozinho os tópicos da álgebra linear. Por último, os exemplos

dados para os alunos podem ter mais conexão com aplicações reais, elevando o seu nível de interesse.

Figura 7 – Proto-persona do Aluno

Infográfico descritivo da proto-persona do aluno

3.2.3 Proto-persona da Programadora

A Programadora, representada na Figura 8, se caracteriza por uma pessoa com conhecimento pelo menos mediano em programação e álgebra linear e que tem interesse em evoluir a biblioteca. A Programadora é ou já foi uma das personas anteriores, a quem agora ajudará na evolução do projeto. Para a Programadora, seu caso de uso é ajudar a melhorar a experiência do aluno e do professor, buscando fazer com que a EasyLinalg se desenvolva e seja mais utilizada.

Figura 8 – Proto-persona da Programadora

Infográfico descritivo da proto-persona da programadora

3.3 MANUAL DE UTILIZAÇÃO

O manual de utilização mostrará de forma prática como utilizar a biblioteca, trazendo exemplos das funcionalidades já implementadas e explicando seus casos de uso. Traz, assim, a visão de como EasyLinalg será utilizada pelas proto-personas de aluno e professor. Já a documentação de implementação utilizada pela proto-persona da programadora, fica a cargo do manual técnico no Capítulo 4.

3.3.1 Instalação

Para instalar a EasyLinalg é necessário o computador já ter previamente instalado o ambiente da Julia. A biblioteca está disponível para instalação através do gerenciador de pacotes da linguagem. Então, na aplicação Júlia acesse o gerenciador de pacotes através da tecla “]” (fechando colchetes) e rode o seguinte comando:

```
add https://github.com/brunohry/Easylinalg
```

Alternativamente, é possível rodar o comando dentro do seu código em Julia, basta utilizar os dois comandos seguintes:

```
using Pkg
```

```
Pkg.add(url="https://github.com/brunohry/Easylinalg")
```

3.3.2 Entrada de dados e Tipos de dados

O fluxo de entrada de dados está diretamente ligado com a tipagem dos mesmos. A interpretação do objeto ocorre na entrada do dado, então o objeto é carregado no tipo de dado específico do seu domínio. Um exemplo prático é o caso de uma imagem. Para carregar essa imagem dentro da EasyLinalg, deve-se utilizar o tipo de dado "*Image*", e então todo o comportamento de imagem implementado pela biblioteca já estará disponível para esse objeto.

Os tipos da EasyLinalg carregam toda a informação de um certo dado, seja ela uma informação semântica, que facilita a interpretação do usuário, ou os metadados que carregam todas as informações necessárias para operar esses tipos. Logo, todos os dados da EasyLinalg devem ser carregados na construção de seus tipos.

Por fim, os construtores dos tipos são as funções de entrada. A seguir serão demonstrados os atuais tipos e como construir objetos com eles.

a) Seta (2D e 3D)
   - Setas, de duas ou três dimensões, são representações vetoriais do $\mathbb{R}^2$ ou $\mathbb{R}^3$ respectivamente, centradas na origem
   - Sua construção é conduzida da seguinte maneira Arrow2D(X, Y) ou Arrow3D(X, Y, Z), onde X, Y e Z são os valores das respectivas coordenadas.

b) Pontos (2D e 3D)
   - Pontos, de duas ou três dimensões, são representações de pontos do $\mathbb{R}^2$ ou $\mathbb{R}^3$ respectivamente
   - Sua construção é conduzida da seguinte maneira Point2D(X, Y) ou Point3D(X, Y, Z), onde X, Y e Z são os valores das respectivas coordenadas.

c) Sinal
   - Sinais são representações de duas dimensões de funções. Podem ser construídos a partir de uma função e um intervalo numérico onde essa função será avaliada, ou através 2 objetos vetoriais, onde um contém as coordenadas do eixo X e o outro, as coordenadas do eixo Y do sinal.
   - Sua construção é conduzida da seguinte maneira:
     SignalBuildU (Função, Inicio:Incremento:Fim) ou
     SignalBuild([X1, X2, X3,. . . ,Xi], [Y1, Y2, Y3,. . . ,Yi])

d) Imagem
   - Imagens são construídas a partir de arquivos de imagens.

– Sua construção é conduzida através da chamada:

Image("caminho/do/arquivo.formato")

A Tabela 1 resume os tipos, seus construtores e as operações básicas definidas para eles.

Tabela 1 – Tipos implementados na EasyLinalg

Tipos	Exemplos de construções	Operações definidas para o tipo	Visualização
Arrow2D	Arrow2D(x1, y1)	+, -, *, /	Draw(Arrow2D(x1, y1))
Arrow3D	Arrow3D(x1, y1, z1)	+, -, *, /	Draw(Arrow2D(x1, y1, z1))
Point2D	Point2D(x1, y1)	+, -, *, /	Draw(Point2D(x1, y1))
Point3D	Point3D(x1, y1, z1)	+, -, *, /	Draw(Arrow2D(x1, y1, z1))
Signal	SignalBuildU(Fx, x1:incremento:xn) e SignalBuild([x1, x2, .., xn] , [y1, y2,...,yn])	+, -, *, /	Draw(Signal(x, y))
Image	Image("/caminho/da/i-magem.formato")	+, -, *, /	Draw(Arrow2D(image1))

Tipos implementados até 04 de setembro de 2022

3.3.3 Operações

Para o escopo inicial da EasyLinalg, foram previstos somente as operações: soma (+), subtração (-), multiplicação por escalar (*), e divisão pela esquerda (\). Sendo que a divisão pela esquerda também é a resolução de um sistema linear.

As operações mencionadas devem ocorrer entre dois objetos de tipos iguais ou entre um objeto de tipo da EasyLinalg e uma matriz numérica ou vetor.

Um resumo dessas operações pode ser visto na Tabela 1. A Tabela 2 detalha melhor estas operações.

Tabela 2 – Operações implementadas na EasyLinalg

Operação	Descrição	Declaração
*	Multiplicação por escalar	Arrow2D(1,2) * 4
*	Multiplicação por matriz	Arrow2D(1,2) * [1 2; 3 4]
/	Divisão entre tipo e escalar	Point2D(8,7) / 3
\	Resolução de sistemas lineares	[2 0;0 3] \ Point2D(4,9)
+	Adição	Point2D(1,2) + Point2D(5,5)
-	Subtração	Point3D(2,2,3) - Point3D(1,2,5)
`convex_combination`	Combinação Convexa	`convex_combination`(Array, step)
RunMarkovChain	Iteração Cadeia de Markov	RunMarkovChain(signal, A, n)

Operações implementados até 04 de setembro de 2022

3.3.4 Saída de dados

A Saída de dados é realizada pela função $Draw(objeto)$. A função Draw aceita dois parâmetros opcionais conforme sua construção abaixo.

```
function Draw(A::types, clear = true, separate = false)
```

Sendo o parâmetro "*clear*" utilizado para limpar a memória temporária de impressão antes de imprimir. Quando "*clear*" é falso, *draws* consecutivos do mesmo tipo devem fazer parte dos mesmos desenhos.

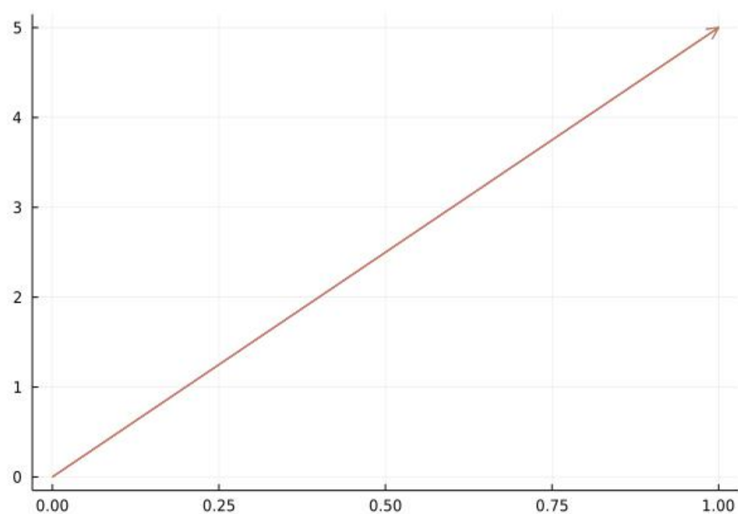
Já o parâmetro "*separate*" serve para indicar se ao receber um vetor, ele deve ser desenhado no mesmo gráfico ou em gráficos separados.

A seguir suas variações e resultados esperados.

a) Seta 2D

- Seta única

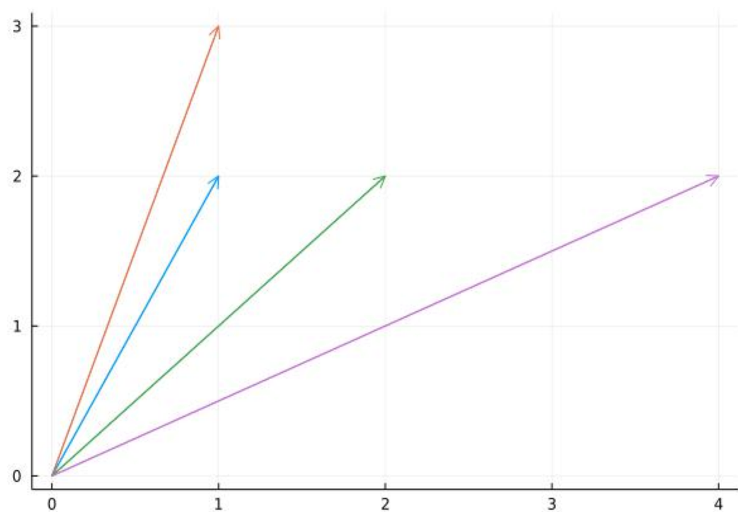
Figura 9 – Draw para seta 2D



Exemplo de saída da função draw

– Array de setas

Figura 10 – Draw para vetor de setas 2D

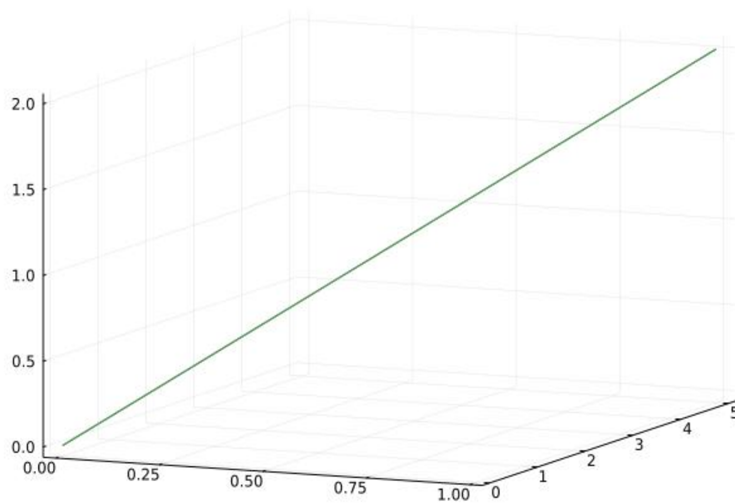


Exemplo de saída da função draw

b) Seta 3D

– Seta única

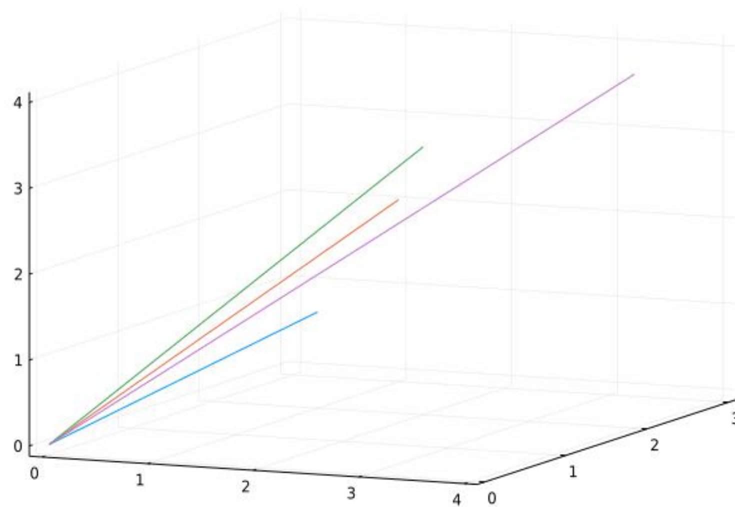
Figura 11 – Draw de seta 3D



Exemplo de saída da função draw

– Array de setas

Figura 12 – Draw para vetor de setas 3D

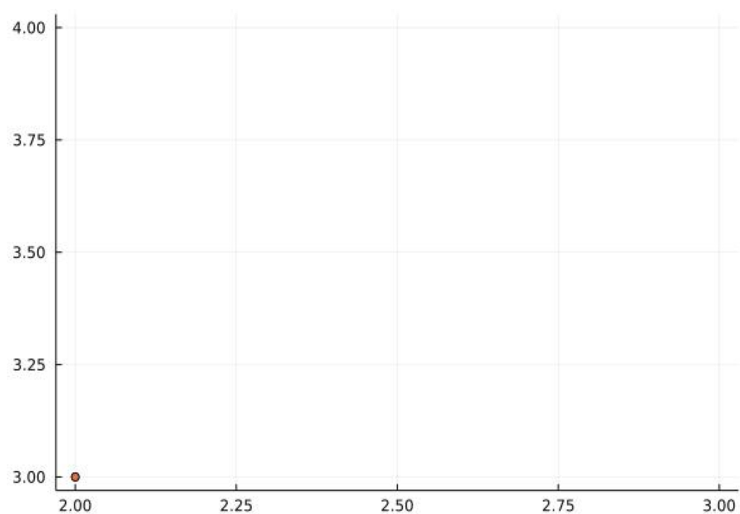


Exemplo de saída da função draw

c) Ponto 2D

– Ponto único

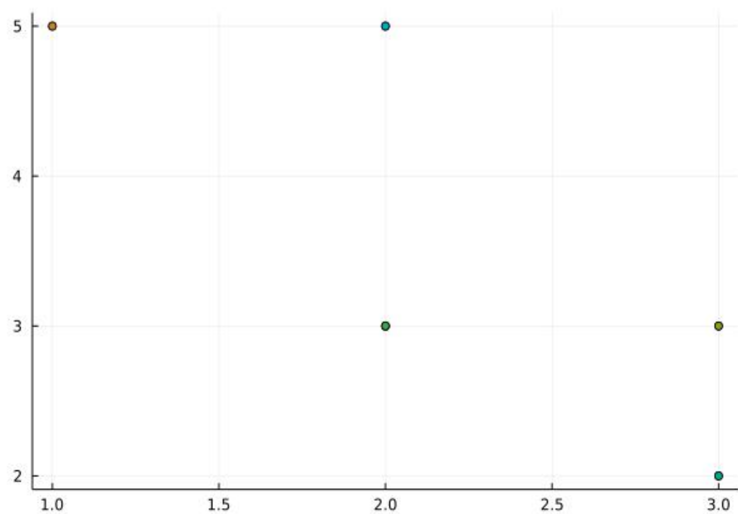
Figura 13 – Draw de ponto 2D



Exemplo de saída da função draw

– Array de pontos

Figura 14 – Draw para vetor de pontos 2D

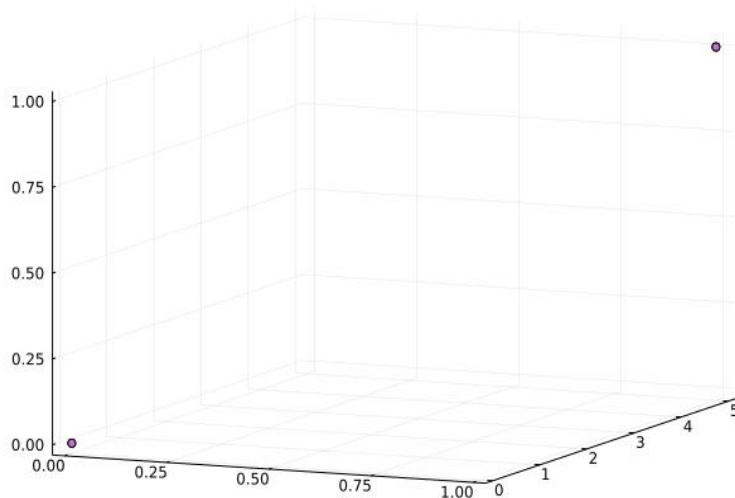


Exemplo de saída da função draw

d) Ponto 3D

– Ponto único

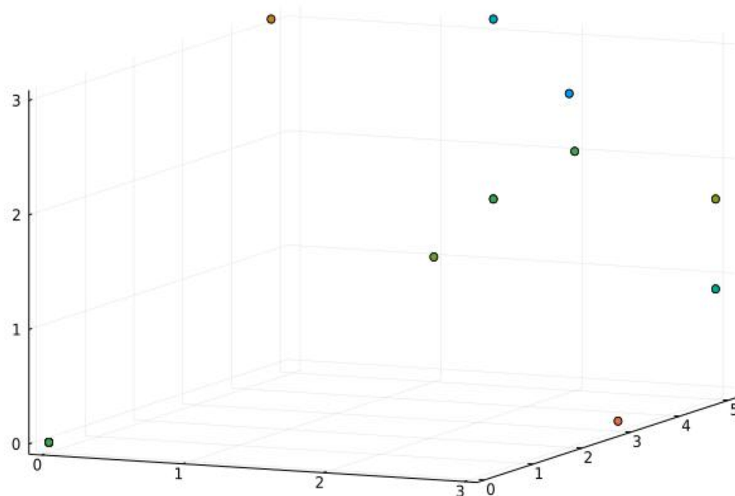
Figura 15 – Draw de ponto 3D



Exemplo de saída da função draw

– Array de pontos

Figura 16 – Draw para vetor de pontos 3D

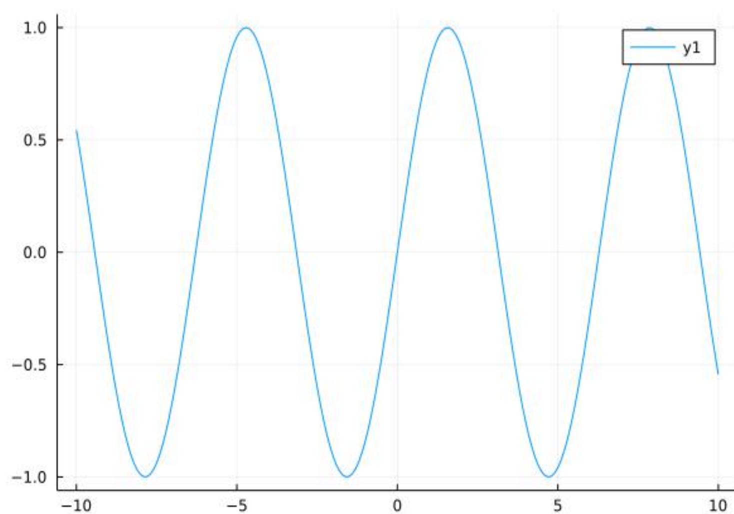


Exemplo de saída da função draw

e) Sinal

– Sinal único

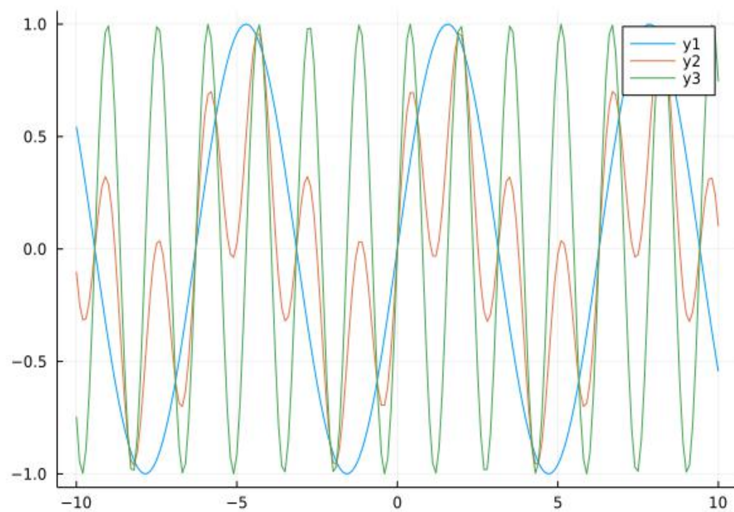
Figura 17 – Draw de um sinal



Exemplo de saída da função draw

– Array de sinais

Figura 18 – Draw para vetor de sinais



Exemplo de saída da função draw

f) Imagem

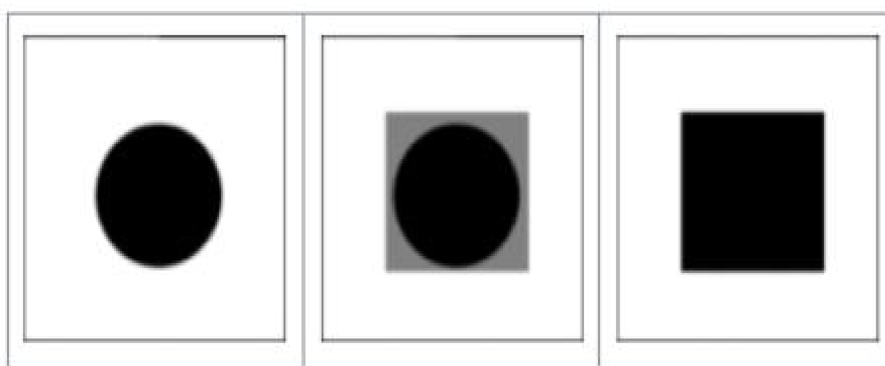
– Imagem única

Figura 19 – Draw de imagem

Exemplo de saída da função draw

– Array de Imagens

Figura 20 – Draw para vetor de imagens



Exemplo de saída da função draw

3.3.5 Aula Exemplo

Nesta sessão será refeita a demonstração da sessão 2.3 dessa vez utilizando a Easy-Linalg, com intuito de mostrar a simplicidade da utilização da biblioteca. O Exemplo abaixo será demonstrado com auxílio da ferramenta Jupyter notebook (JUPYTER, s.d.) mostrando sempre o código e o resultado de sua execução. Mais exemplos podem ser encontrados na sessão 3.4.

Ao explicar uma transformação linear de reflexão, por exemplo, podemos dizer que a matriz mathmode A

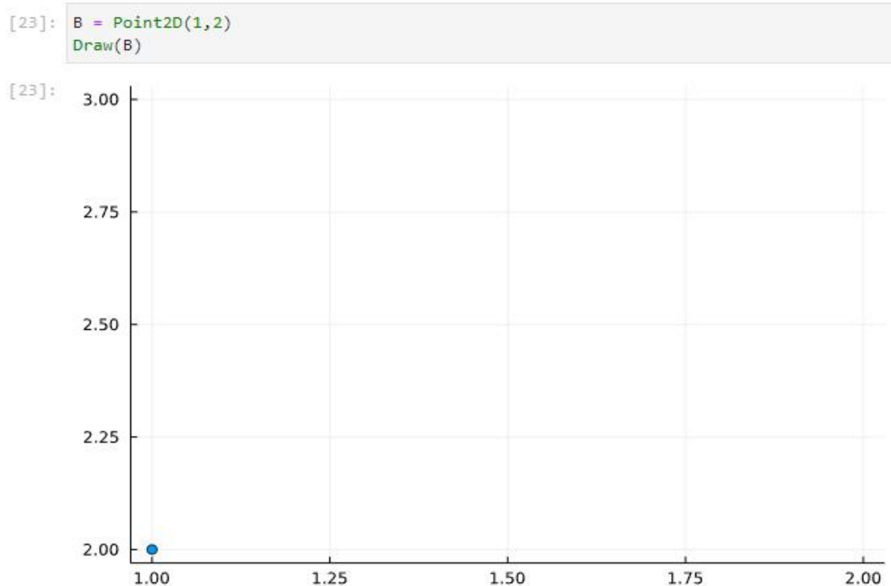
Figura 21 – Matriz A

```
[22]: using EasyLinalg
      A = [1 0;
           0 -1]

[22]: 2x2 Matrix{Int64}:
       1   0
       0  -1
```

Matriz de reflexão no eixo X escrita na EasyLinalg

espelha um ponto do $\mathbb{R}^2$ ao redor do eixo X, ou seja, se for multiplicado um ponto do $\mathbb{R}^2$ por essa matriz, ele terá sua coordenada Y invertida. Usando como exemplo o ponto B:

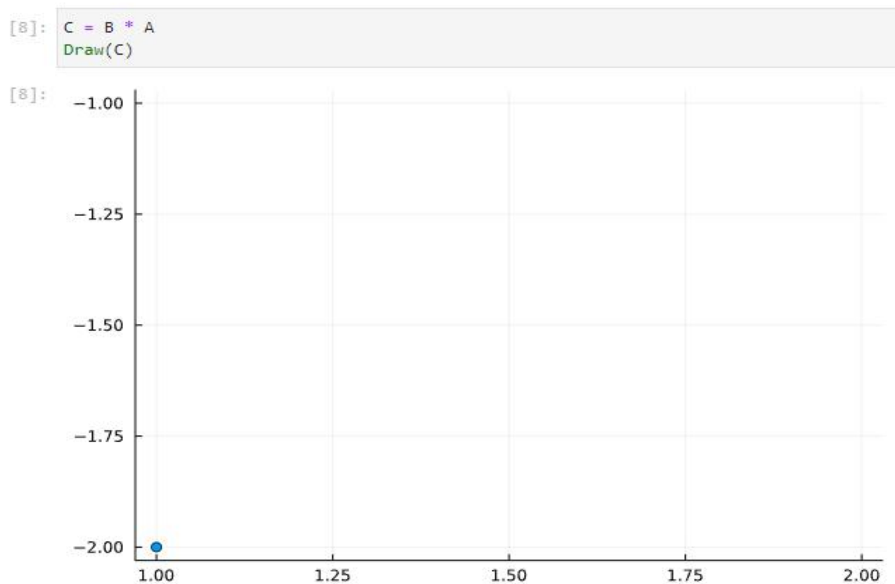
Figura 22 – Ponto B



Ponto (1,2) escrito na EasyLianlg

Podemos aplicar a transformação de reflexão:

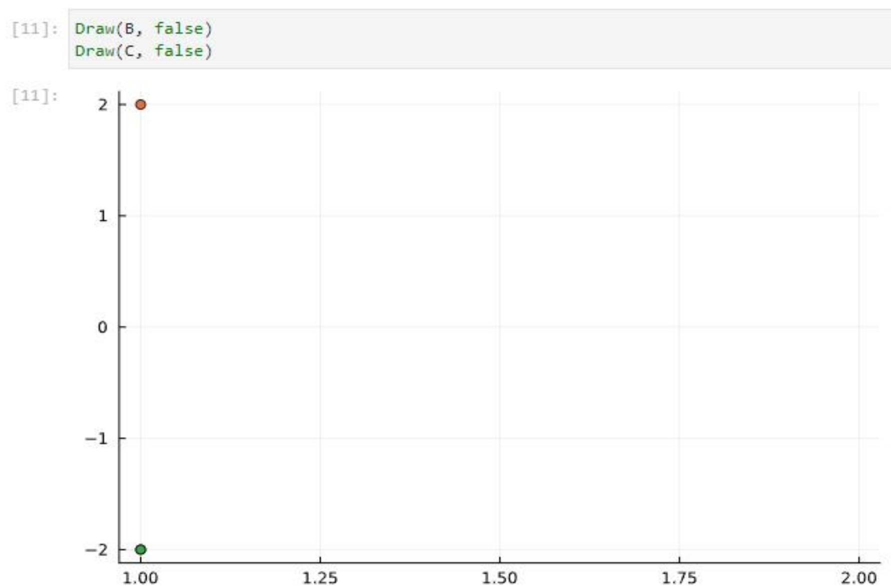
Figura 23 – Ponto C



Reflexão do ponto B escrito na EasyLianlg

Por fim, para facilitar a visualização, desenharemos os dois pontos no mesmo gráfico:

Figura 24 – Ponto B e Ponto C

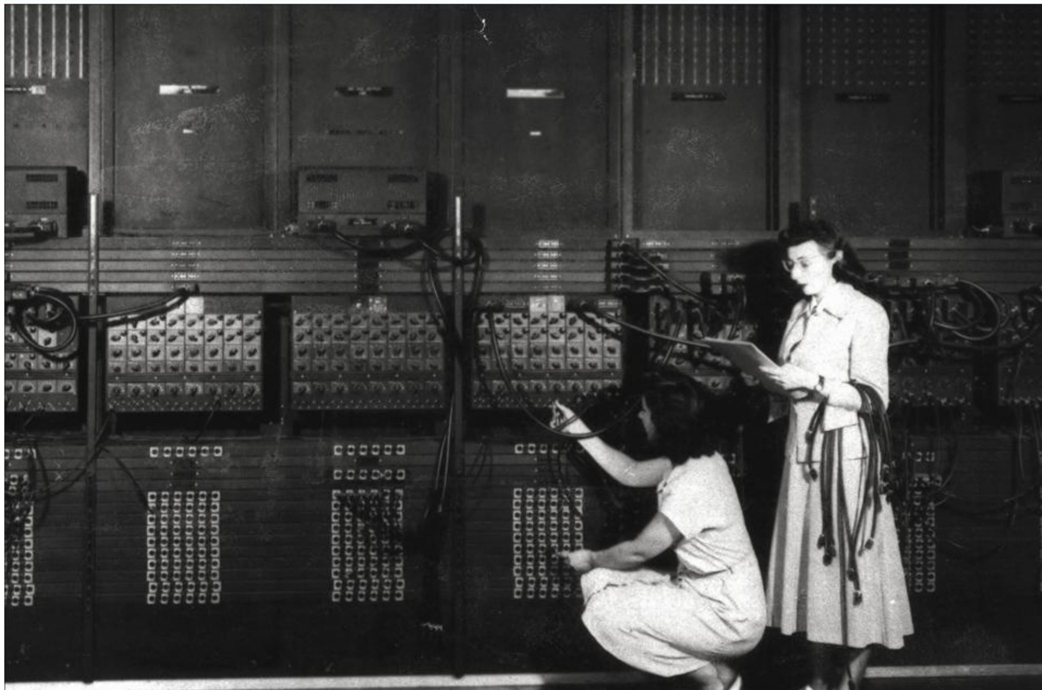


Ponto B e Ponto C escrito na EasyLianlg

Agora, para melhor exemplificar, podemos ilustrar onde isso pode ser aplicado na computação. Para isso utilizaremos a imagem:

Figura 25 – Imagem e Código

```
[24]: eniac = Image("eniac.jpg")
      Draw(eniac)
```



Entrada e saída da imagem Eniac na EasyLianlg

Aplicando então a matriz A (Figura 21), teremos um espelhamento em torno do eixo Y.

Figura 26 – Imagem e Código

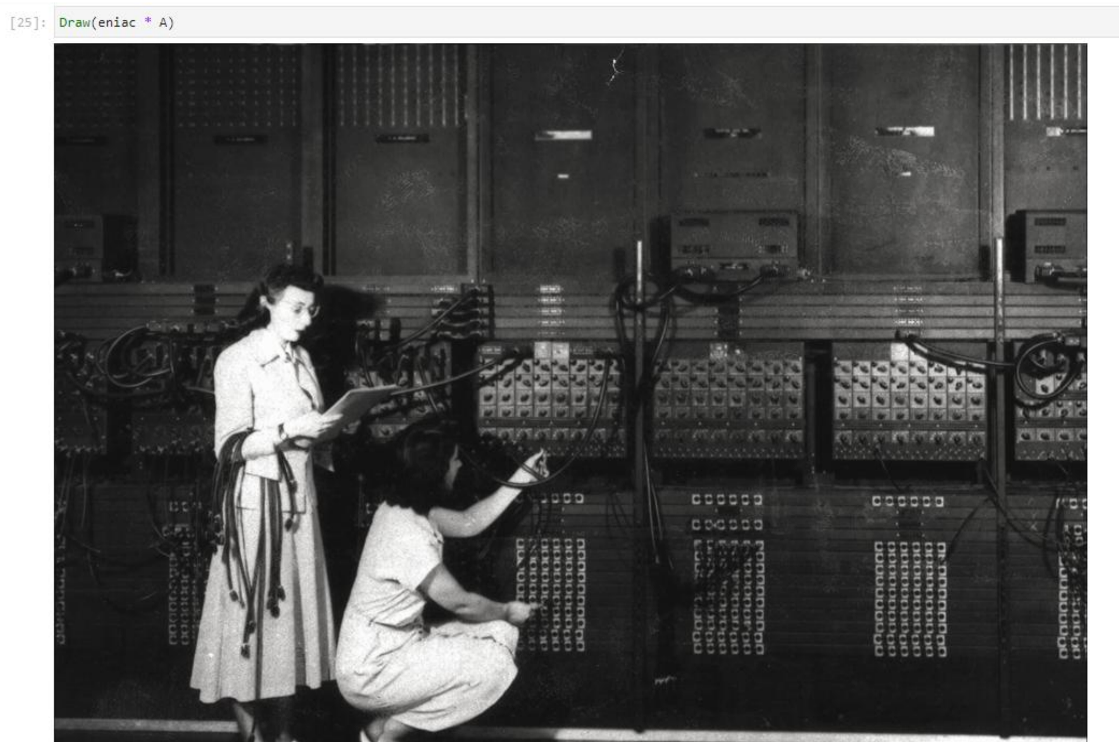


Imagem Eniac refletida escrita na EasyLianlg

Por fim, vale lembrar que a reflexão ocorreu no eixo Y e não X devido à padronização dos píxeis já descrita na subseção 2.3.3

Ao terminar o exemplo, pode-se então notar a simplicidade da construção do exemplo tanto com um ponto quando com a imagem, onde mudando apenas a função de entrada, foi possível reproduzir o mesmo resultado para objetos de naturezas distintas.

3.4 RESULTADOS

Este trabalho visou diminuir a distância entre matemática e computação em sala de aula focando em exemplificações de álgebra linear. Na sessão a seguir, serão demonstrados alguns resultados do trabalho, que essencialmente são exemplificações de conceitos de álgebra linear elaborados através da biblioteca. Serão exemplificadas 5 aplicações as quais servirão de pretexto para demonstrar as operações construídas, são elas: média aritmética de objetos, transição de um objeto em outro (*morphing*), sistemas lineares, transformações lineares e cadeias de Markov.

Um ponto a se observar é como o mesmo exemplo utiliza sempre o mesmo código mudando apenas os tipos de entrada, conseguindo adaptar e entregar os entregar resultados esperados sem intervenção para cada tipo de entrada.

Por último, é importante ressaltar que, os resultados ilustrados nas próximas páginas

foram desenvolvidos utilizando a biblioteca junto ao ambiente de desenvolvimento integrado Jupyter Notebook, e seus respectivos códigos fonte se encontram na pasta 'docs/Exemplos' no repositório público do projeto no Github.

3.4.1 Exemplo de multiplicação por escalar e soma: média aritmética

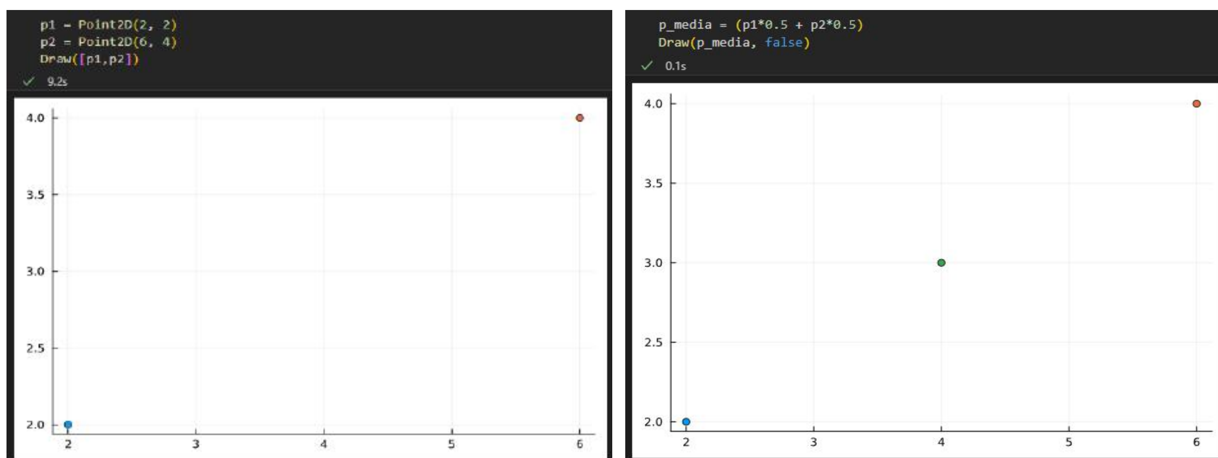
A fim de demonstrar a utilização das operações de multiplicação por escalar e soma entre tipos, utilizaremos o exemplo de média aritmética entre dois objetos de mesmo tipo da EasyLinalg.

A média aritmética entre objetos pode ser definida como $(N1 * (1/i) + N2 * (1/i) + ... + Ni * (1/i))$, e representa a concentração de um conjunto de objetos. Podemos então usar a EasyLinalg para exemplificar este comportamento.

a) Pontos

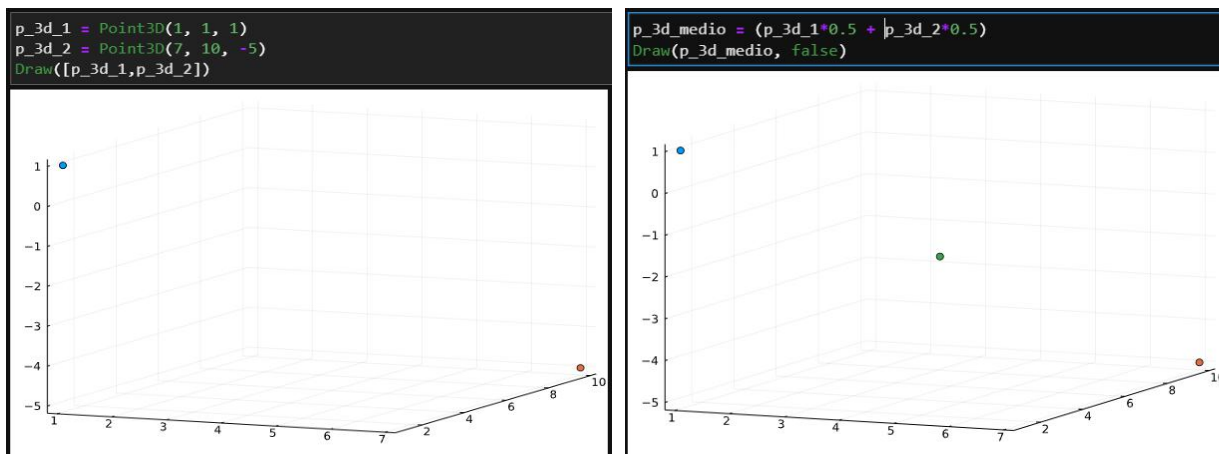
Definidos dois pontos, são então desenhados, conforme observado no lado esquerdo da Figura 27 e Figura 28. A partir desses pontos é calculada a média. Na direita da Figura 27 e Figura 28 vemos então o cálculo da média e o novo ponto desenhado exatamente no meio dos dois pontos originais.

Figura 27 – Exemplo de média aritmética de Pontos 2D



À esquerda o conjunto inicial; à direita o conjunto inicial e o resultado do exemplo

Figura 28 – Exemplo de média aritmética de Pontos 3D

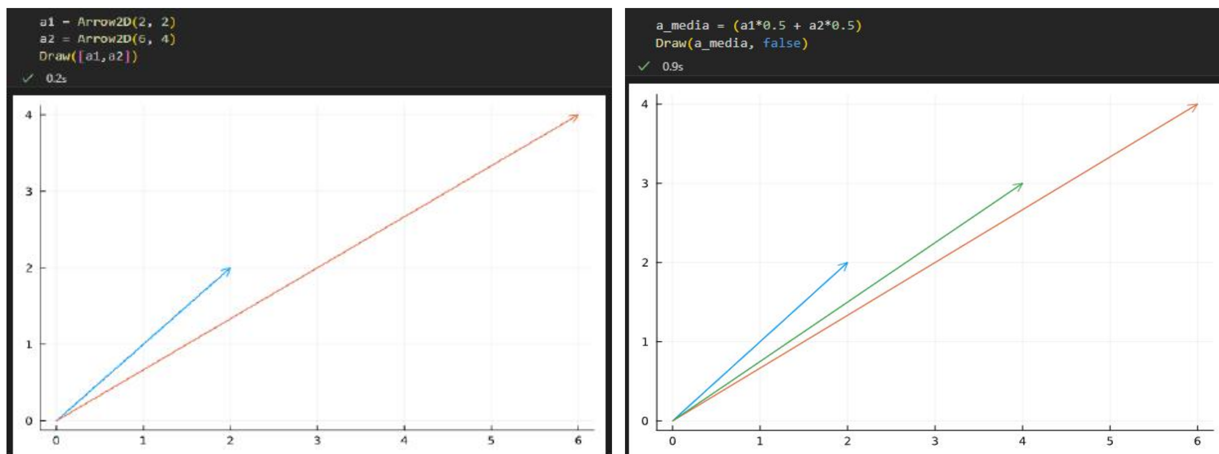


À esquerda o conjunto inicial; à direita o conjunto inicial e o resultado do exemplo

b) Setas

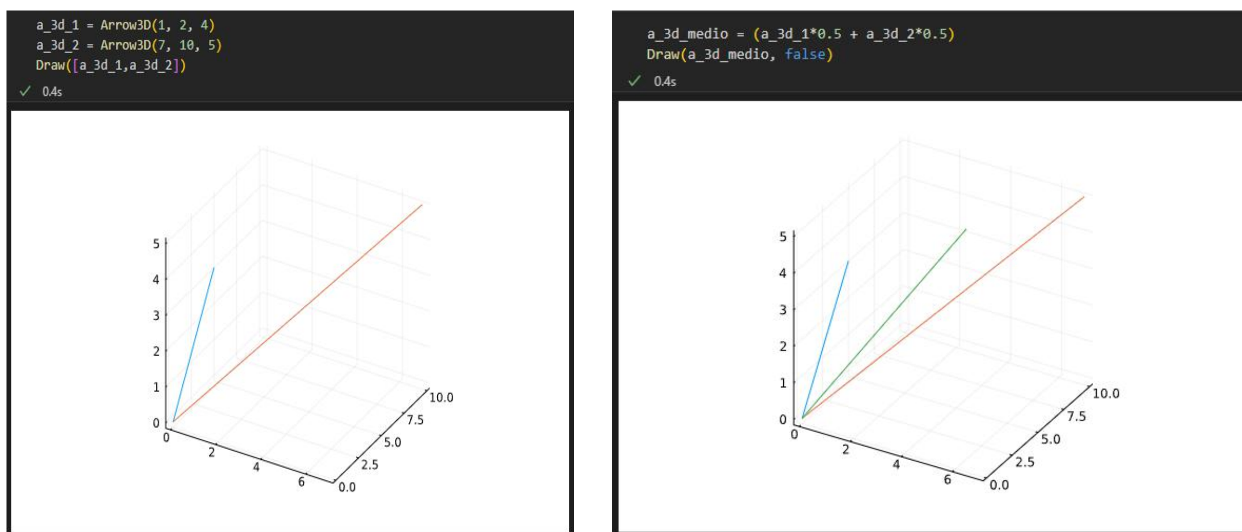
Definidas duas setas, são então desenhadas, conforme observado no lado esquerdo da Figura 29 e Figura 30. A partir dessas setas é calculada a média. Na direita da Figura 29 e Figura 30 vemos então o cálculo da média e a nova seta desenhada exatamente no meio das duas setas originais.

Figura 29 – Exemplo de média aritmética de Setas 2D



À esquerda o conjunto inicial; à direita o conjunto inicial e o resultado do exemplo

Figura 30 – Exemplo de média aritmética de Setas 3D

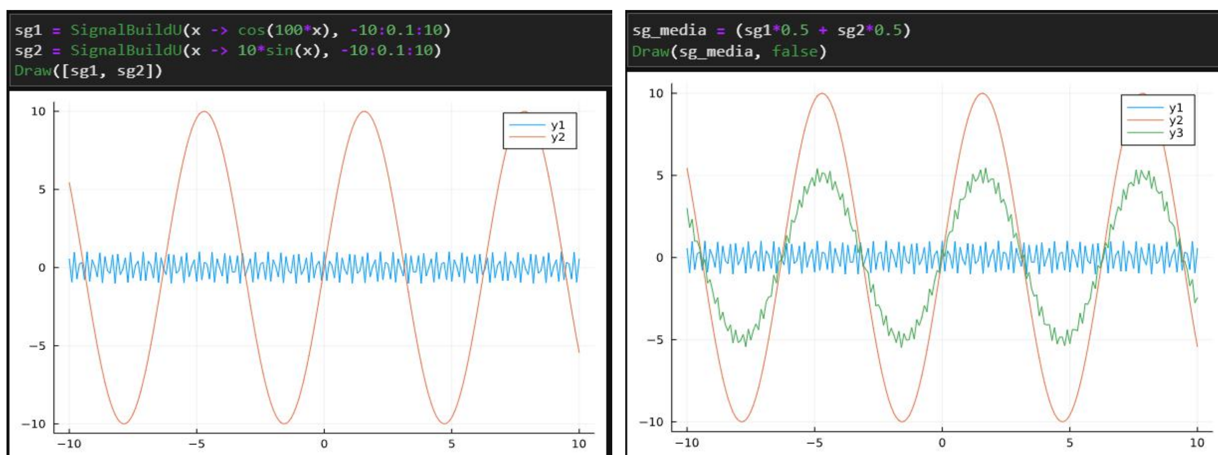


À esquerda o conjunto inicial; à direita o conjunto inicial e o resultado do exemplo

c) Sinais

Definidos dois sinais, são então desenhados, conforme observado no lado esquerdo da Figura 31. A partir desses sinais é calculada a média. Na direita da Figura 31 vemos então o cálculo da média e o novo sinal desenhado junto aos sinais originais.

Figura 31 – Exemplo de média aritmética de Sinais



A esquerda o conjunto inicial, a direita o conjunto inicial e o resultado do exemplo

d) Imagens

Definidas duas imagens, são então desenhadas, conforme observado no lado esquerdo da Figura 32. A partir dessas imagens é calculada a média. Na direita da Figura 32 vemos então o cálculo da média e a nova imagem.

Figura 32 – Exemplo de média aritmética de Imagens

À esquerda o conjunto inicial, à direita o resultado do exemplo

3.4.2 Exemplo combinação convexa: morphing

O termo *morphing* consiste na transição de um objeto em outro. Esta transformação começa com 100% do primeiro objeto e, a cada iteração, diminui essa porcentagem do primeiro objeto e adiciona a mesma proporção do segundo objeto, terminando então com 100% do segundo objeto. A biblioteca apresenta essa função implementada através da chamada:

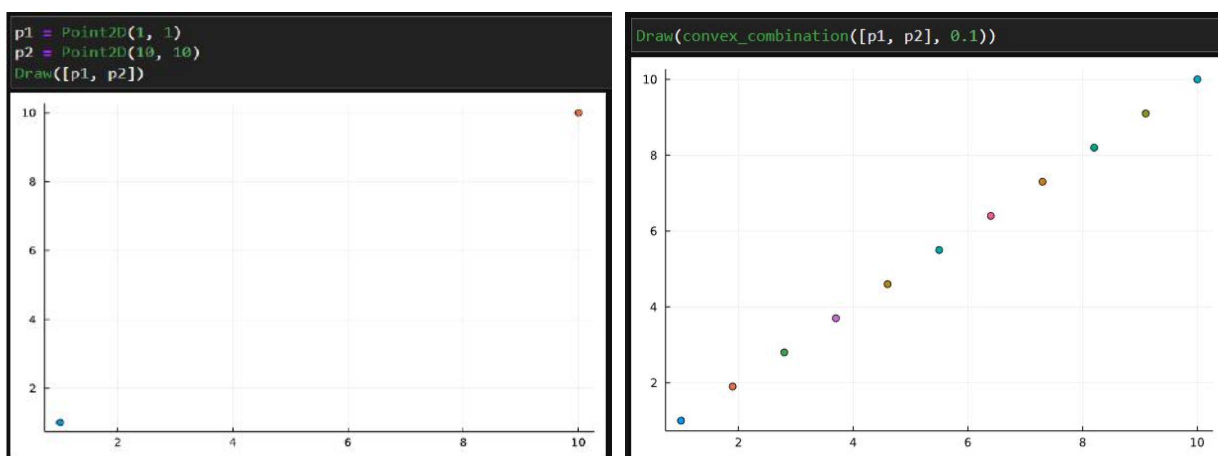
```
convex_combination(vetor_de_entrada, incremento)
```

Onde o vetor de entrada é um vetor bidimensional que contém os objetos que devem ser transformados, sendo o da primeira posição o objeto inicial e o da segunda, o objeto final. Já o incremento é um valor entre 0 e 1 que representa a porcentagem de incremento de cada iteração.

a) Pontos

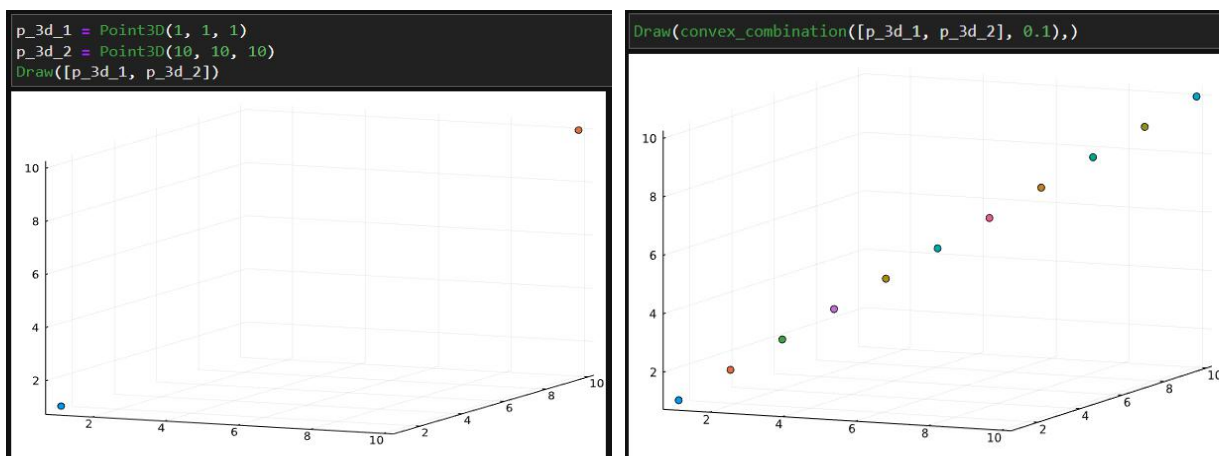
Definidos dois pontos, são então desenhados, conforme observado no lado esquerdo da Figura 33 e Figura 34. A partir desses pontos é aplicada a combinação convexa, na direita da Figura 33 e Figura 34 vemos esta aplicação e o conjunto resultado. O conjunto resultado mostra todas as iterações, onde cada iteração gera um novo ponto que se distancia do primeiro e se aproxima do segundo.

Figura 33 – Exemplo de morphing para Pontos 2D



À esquerda o conjunto inicial, à direita o conjunto inicial e o resultado do exemplo

Figura 34 – Exemplo de morphing para Pontos 3D

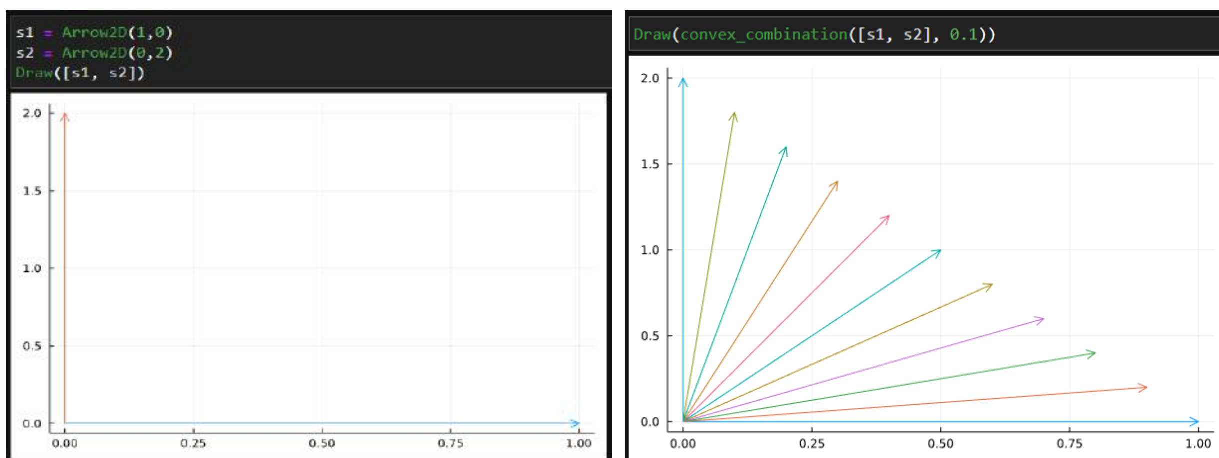


À esquerda o conjunto inicial, à direita o conjunto inicial e o resultado do exemplo

b) Setas

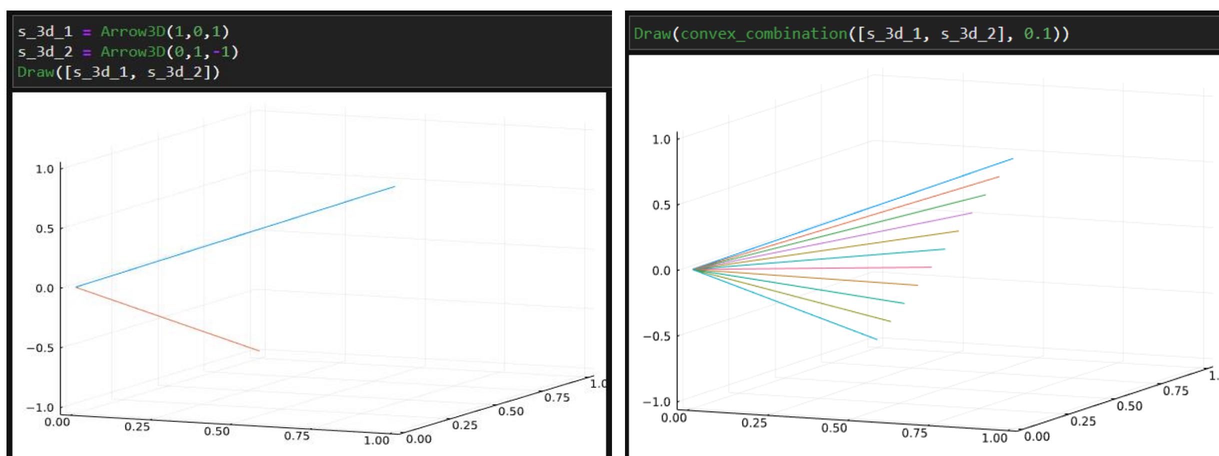
Definidas duas setas, são então desenhadas, conforme observado no lado esquerdo da Figura 35 e Figura 36. A partir dessas setas é aplicada a combinação convexa, na direita da Figura 35 e Figura 36 vemos esta aplicação e o conjunto resultado. O conjunto resultado mostra todas as iterações, onde cada iteração gera uma nova seta que se distancia da primeira e se aproxima da segunda.

Figura 35 – Exemplo de morphing para Setas 2D



À esquerda o conjunto inicial, à direita o conjunto inicial e o resultado do exemplo

Figura 36 – Exemplo de morphing para Setas 3D

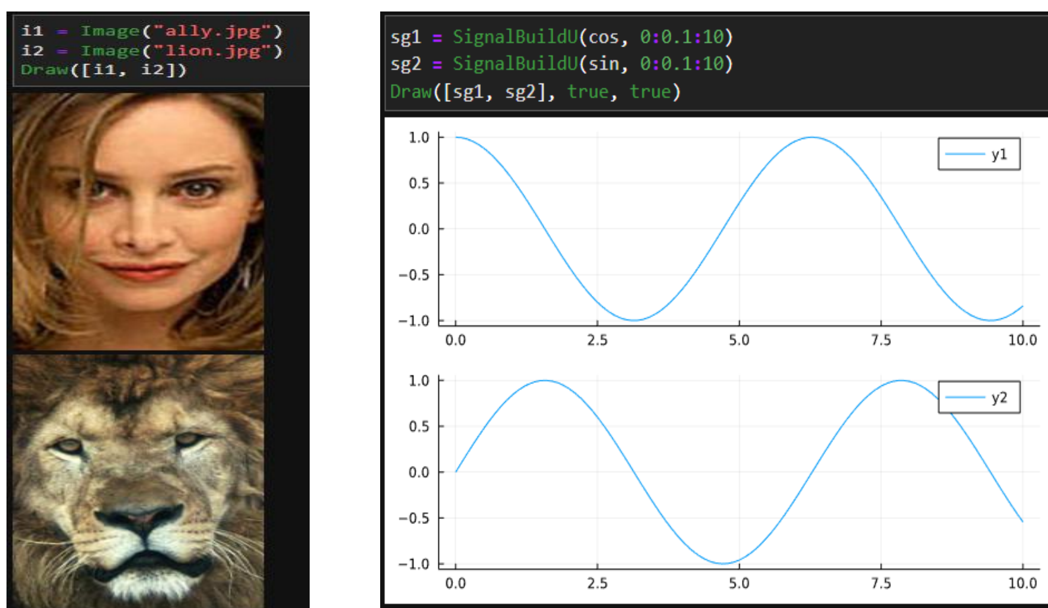


À esquerda o conjunto inicial, à direita o conjunto inicial e o resultado do exemplo

c) Imagens e Sinais

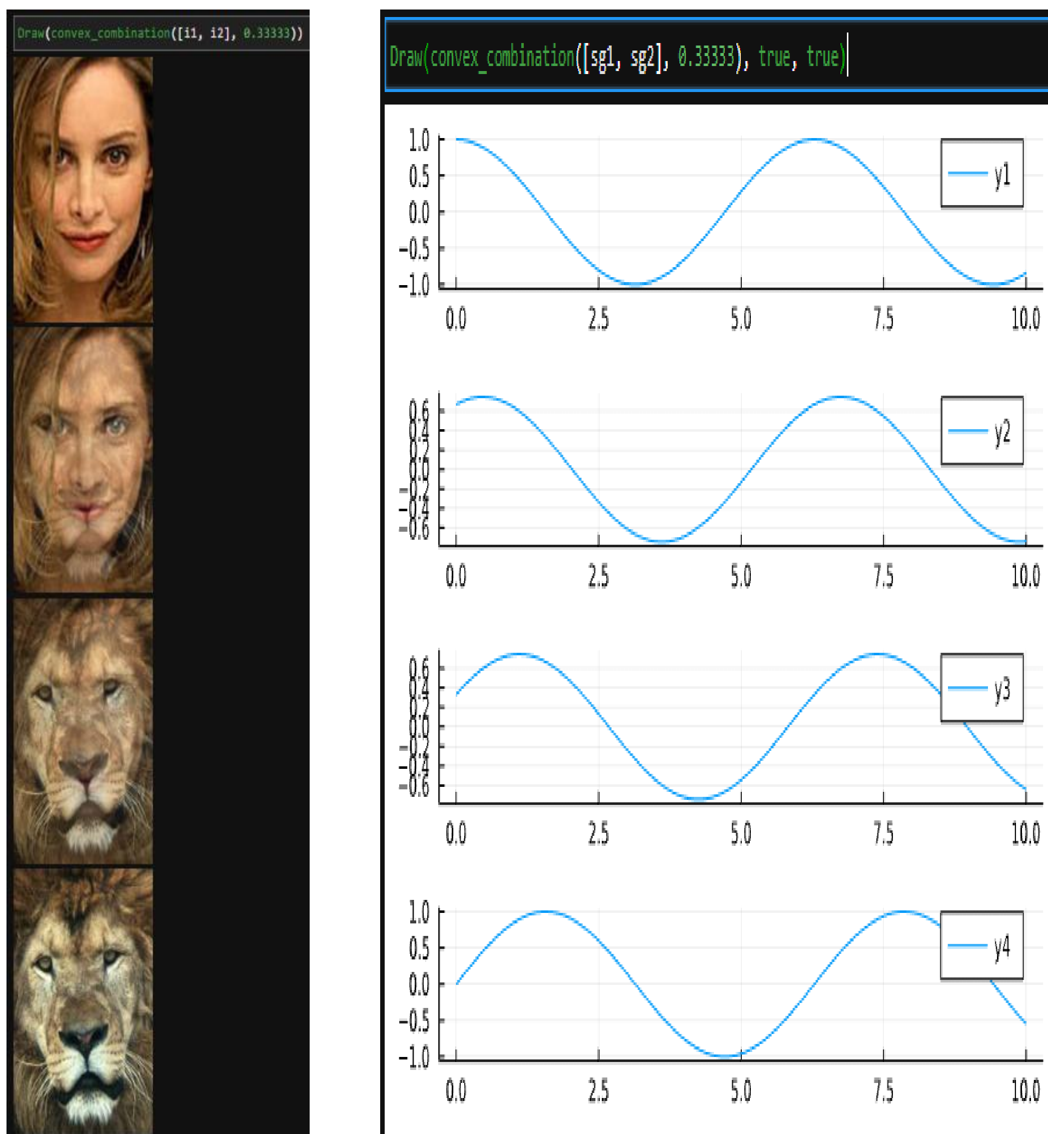
As imagens e os sinais são alinhados nesse exemplo para facilitar a visualização dos efeitos, então na Figura 37 temos definidos as duas imagens iniciais e os dois sinais iniciais. Ao aplicar a combinação convexa, na Figura 38, é possível observar que a cada iteração o primeiro objeto se distancia de sua forma original e se aproxima então da forma do objeto final.

Figura 37 – Imagens e Sinais para exemplo de morphing



As imagens e sinais que serão referências de primeiro e último estágio de morphing

Figura 38 – Morphing de Imagens e Sinais



Resultado do *morphing*

3.4.3 Exemplo de multiplicação por Matriz: transformações lineares

Uma transformação linear é uma operação que move um objeto entre dois espaços vetoriais. Sua aplicação é muito importante no âmbito da computação gráfica, onde é muito utilizada para manipular imagens. Basicamente uma transformação linear é dada por

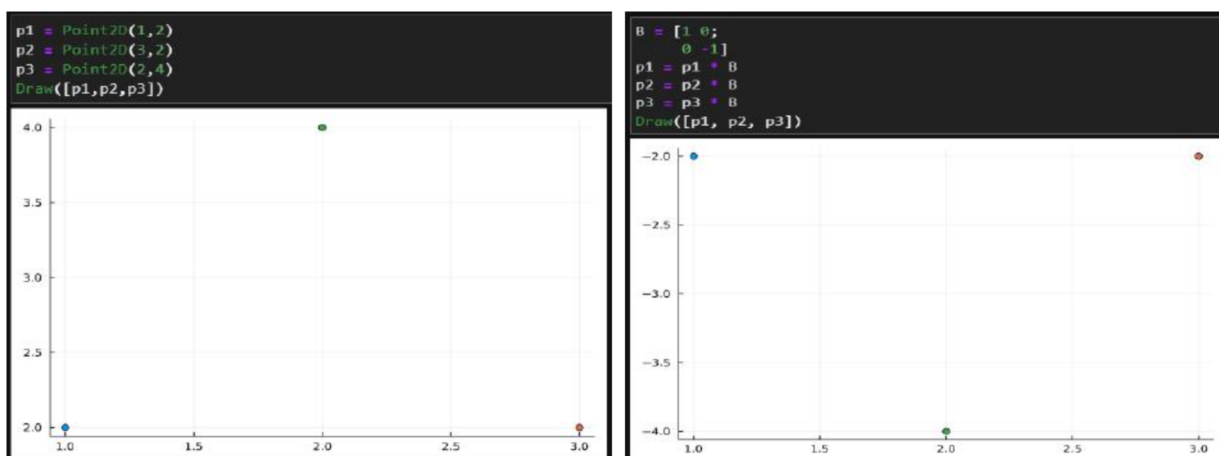
```
A x B = C
```

Onde A é o objeto de origem, B é a matriz que transforma esse objeto e C o resultado. Abaixo serão demonstrados exemplos dessas transformações com a EasyLinalg.

a) Pontos

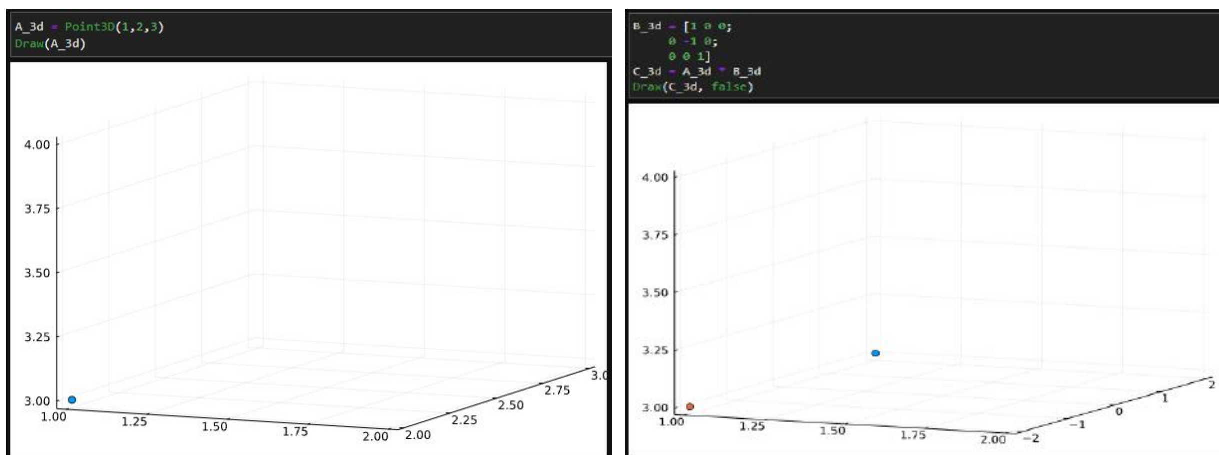
Definidos os pontos iniciais, são então desenhados, conforme observado no lado esquerdo da Figura 39 e Figura 40. A partir desses pontos é aplicada a transformação linear de reflexão no eixo X. Na direita da Figura 39 e Figura 40 vemos a aplicação da transformação e o conjunto resultado.

Figura 39 – Exemplo de transformação linear de reflexão para Pontos 2D



À esquerda o conjunto inicial, à direita o resultado

Figura 40 – Exemplo de transformação linear de reflexão para Pontos 3D

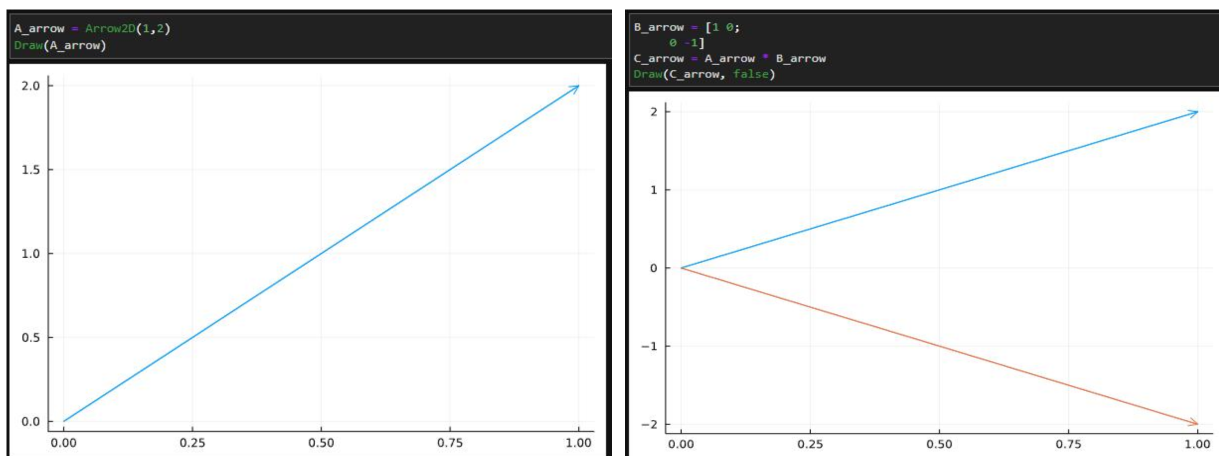


À esquerda o conjunto inicial, à direita o conjunto inicial e o resultado do exemplo

b) Setas

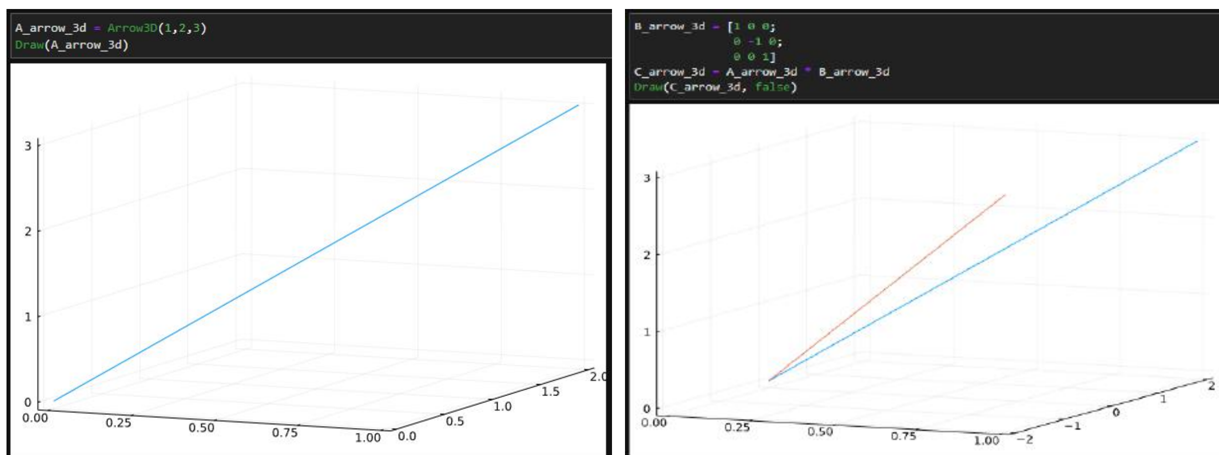
Definida a seta, é então desenhada, conforme observado no lado esquerdo da Figura 41 e Figura 42. A partir dessa seta é aplicada a transformação linear de reflexão no eixo X. Na direita da Figura 41 e Figura 42 vemos a aplicação da transformação e a seta refletida ao lado da original.

Figura 41 – Exemplo de transformação linear de reflexão para Setas 2D



À esquerda o conjunto inicial, à direita o conjunto inicial e o resultado do exemplo

Figura 42 – Exemplo de transformação linear de reflexão para Setas 3D



À esquerda o conjunto inicial, à direita o conjunto inicial e o resultado do exemplo

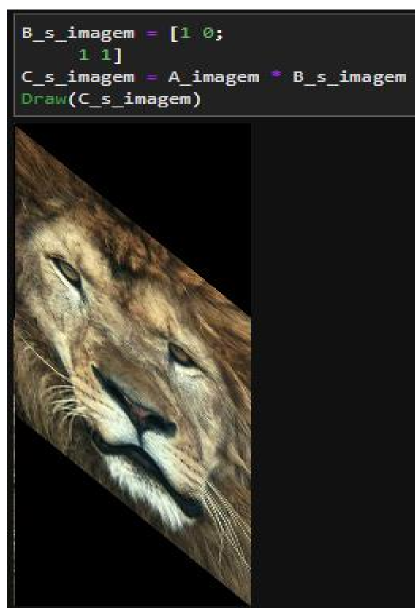
c) Imagens

Definida a imagem original é então desenhada, conforme observado no lado esquerdo da Figura 43, a partir disso é então aplicada a transformação. Esta transformação é aplicada píxel a píxel em cima de suas coordenadas, resultando na imagem a direita da Figura 43. Por fim, a Figura 44 mostra a aplicação de transformação de cisalhamento, a fim dar outra perspectiva do funcionamento das transformações lineares em imagens.

Figura 43 – Exemplo de transformação linear de reflexão para Imagens

À esquerda imagem inicial, à direita o resultado

Figura 44 – Transformação linear de cisalhamento



Resultado da transformação linear de cisalhamento

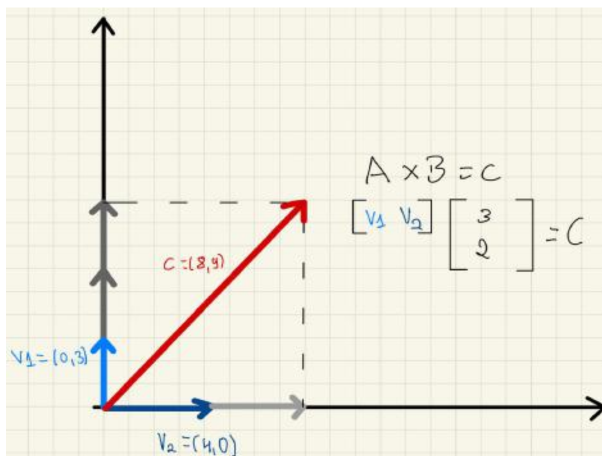
3.4.4 Exemplo de resolução de sistemas lineares

A resolução de sistemas lineares pode ser utilizada para decompor objetos como demonstrado na Figura 45 onde $C = 3v_1 + 2v_2$. Para exemplificar com os objetos da EasyLinalg, utilizaremos o seguinte produto vetorial

```
A x B = C
```

Onde A é um vetor coluna de tipos da EasyLinalg e B um vetor coluna de tipos numéricos, o resultado C é uma soma dos itens de A ponderados por B, mantendo o tipo dos itens de A.

Figura 45 – Exemplo A X B = C



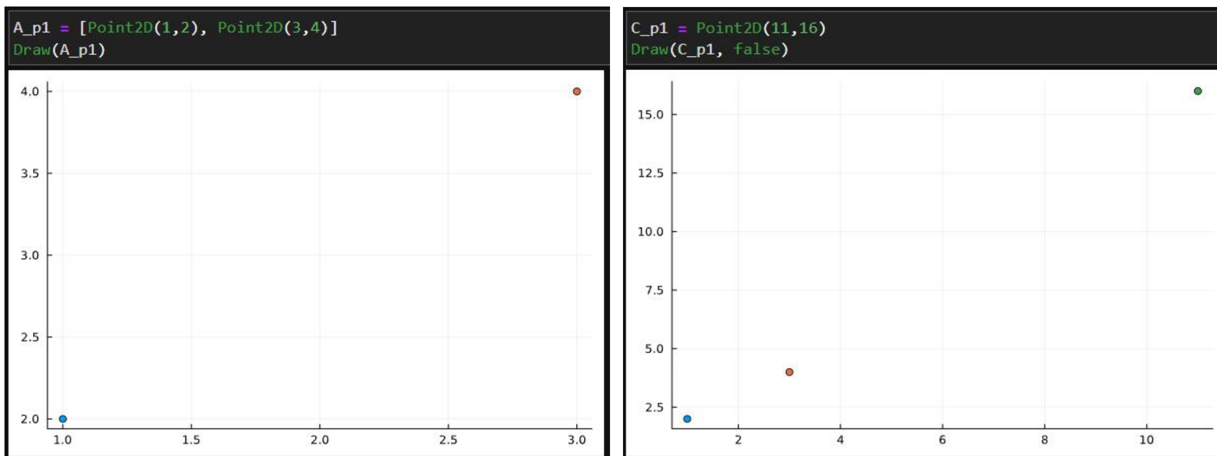
Exemplo de A X B = C

Podemos então aplicar o operador contra-barra de Julia para resolver o problema de achar B tendo A e C, assim descobrindo a porção que cada elemento de A contribui em C. Note então que este exemplo é a operação contrária a operação do exemplo anterior.

a) Pontos 2D

Definido o vetor de dois pontos que representa A, é desenhado do lado esquerdo da Figura 46. Então é definido o ponto C, desenhado junto ao o vetor A no lado direito da Figura 46. Na Figura 47 é então calculado o sistema linear onde sua resolução é dada pelo vetor resultado.

Figura 46 – Componentes A e C



Componentes A e C para exemplo de decomposição de Pontos 2D

Figura 47 – Componente B, resultado para exemplo de decomposição de Pontos 2D

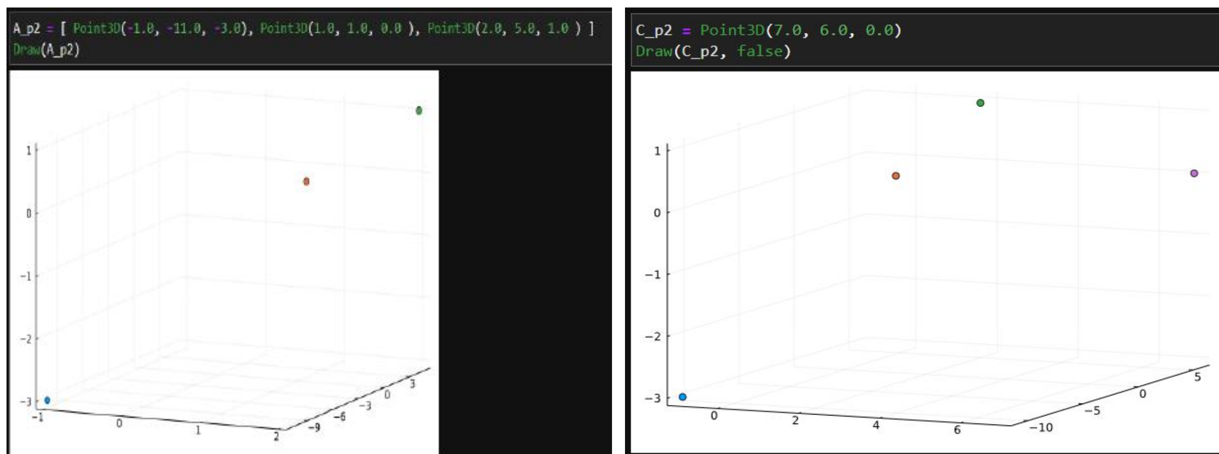
```
B_p1 = A_p1\C_p1

2-element Vector{Float64}:
 2.0
 3.0
```

Vetor de resultado

b) Pontos 3D

Definido o vetor de três pontos que representa A, é desenhado do lado esquerdo da Figura 48. Então é definido o ponto C, desenhado junto ao o vetor A no lado direito da Figura 48. Na Figura 49 é então calculado o sistema linear onde sua resolução é dada pelo vetor resultado.

Figura 48 – Componentes A e C



Componentes A e C para exemplo de decomposição de Pontos 3D

Figura 49 – Componente B, resultado para exemplo de decomposição de Pontos 3D

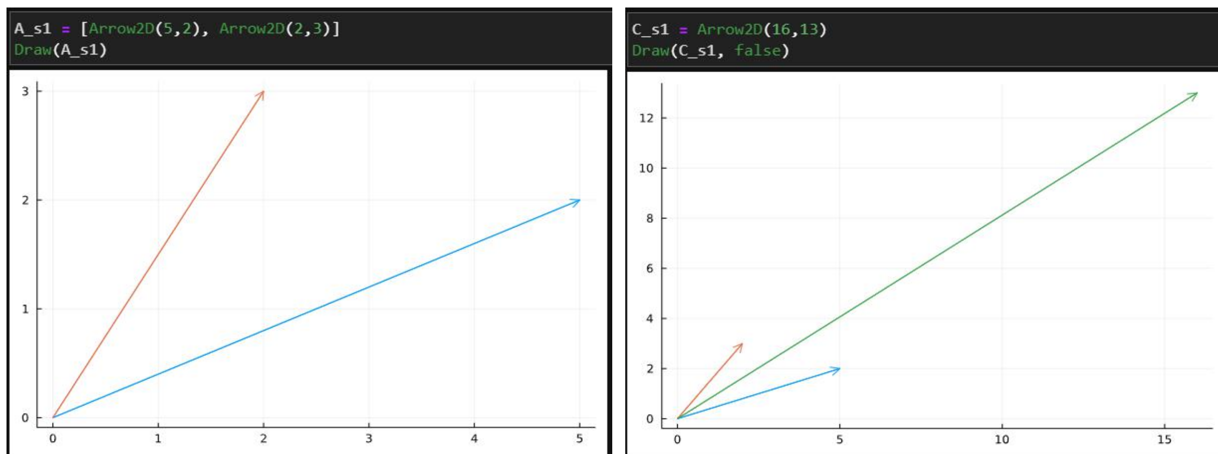
```
B_p2 = A_p2\C_p2

3-element Vector{Float64}:
 1.0
 2.0000000000000004
 3.0
```

Vetor de resultado

c) Setas 2D

Definido o vetor de duas setas que representa A, é desenhado do lado esquerdo da Figura 50. Então é definida a seta C, desenhada junto ao o vetor A no lado direito da Figura 50. Na Figura 51 é então calculado o sistema linear onde sua resolução é dada pelo vetor resultado.

Figura 50 – Componentes A e C



Componentes A e C para exemplo de decomposição de Setas 2D

Figura 51 – Componente B, resultado para exemplo de decomposição de Setas 2D

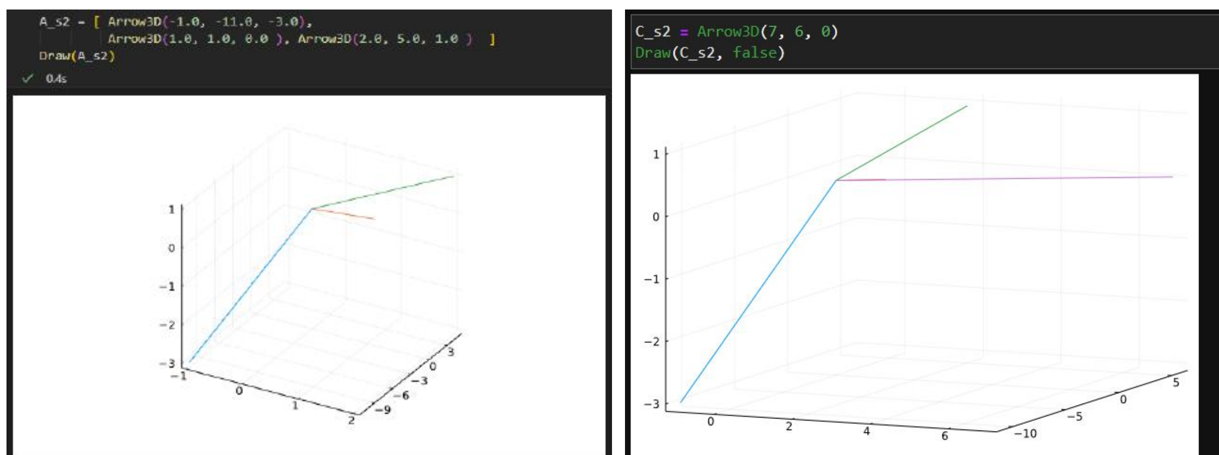
```
B_s1 = A_s1\C_s1

2-element Vector{Float64}:
 2.0
 2.9999999999999996
```

Vetor de resultado

d) Setas 3D

Definido o vetor de três setas que representa A, é desenhado do lado esquerdo da Figura 52. Então é definida a seta C, desenhada junto ao o vetor A no lado direito da Figura 52. Na Figura 53 é então calculado o sistema linear onde sua resolução é dada pelo vetor resultado.

Figura 52 – Componentes A e C



Componentes A e C para exemplo de decomposição de Seta 3D

Figura 53 – Componente B, resultado para exemplo de decomposição de Setas 3D

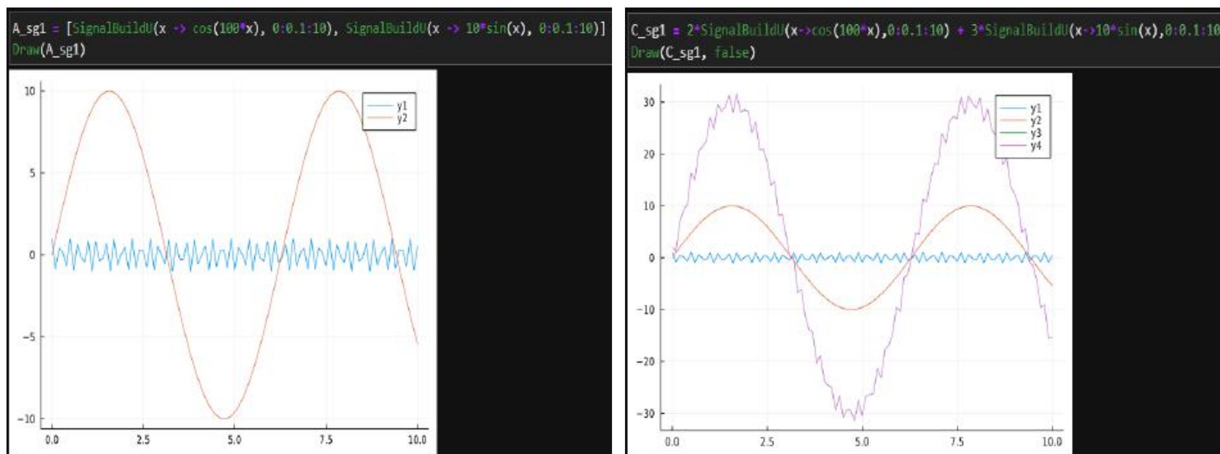
```
B_s2 = A_s2\C_s2

3-element Vector{Float64}:
 1.0
 2.0000000000000004
 3.0
```

Vetor de resultado

e) Sinais

Definido o vetor de dois sinais que representa A, é desenhado do lado esquerdo da Figura 54. Então é definido o sinal C, desenhado junto ao o vetor A no lado direito da Figura 54. Na Figura 55 é então calculado o sistema linear onde sua resolução é dada pelo vetor resultado.

Figura 54 – Componentes A e C



Componentes A e C para exemplo de decomposição de Sinais

Figura 55 – Componente B, resultado para exemplo de decomposição de Sinais

```
B_sg1 = A_sg1\C_sg1

2-element Vector{Float64}:
 2.000000000000001
 3.0
```

Vetor de resultado

3.4.5 Exemplo de cadeias de Markov

A cadeia de Markov é uma representação algébrica de transição de estados. Imagine uma máquina de estados finitos e um passeio aleatório pelos estados, onde as probabilidades de transição de um estado para o outro são conhecidas. Desta forma, independente de quantos passos sejam dados, será apenas necessário o estado atual para saber a probabilidade de ir para qualquer outro estado do sistema. A representação da cadeia é utilizada para se obter o estado de equilíbrio do sistema, que se baseia na convergência da matriz de transição de estados. A cadeia de Markov é utilizada para interpretar dados estatísticos como ranking de sites, previsão de valor de um ativo na bolsa de valores, previsão de acionamentos de sinistros de seguros, entre outros.

A biblioteca aqui apresentada mostra visualmente a convergência de um dado tipo de entrada, e.g. seta 2D, até seu estado de equilíbrio. Observemos a equação a seguir:

$$A_{mxm} * P_m$$

Onde:

- A é uma matriz estocástica de transição de estados, $m \times m$, onde m é o tamanho do tipo que será utilizado
- P é um tipo implementado na biblioteca

A cada iteração da cadeia até a convergência, teremos uma nova potência da matriz A sendo multiplicada por um vetor em sua atual iteração. Eventualmente, o vetor para de variar, chegando no nosso objetivo.

a) Pontos 2D

Figura 56 – Cadeia de Markov em Pontos 2D

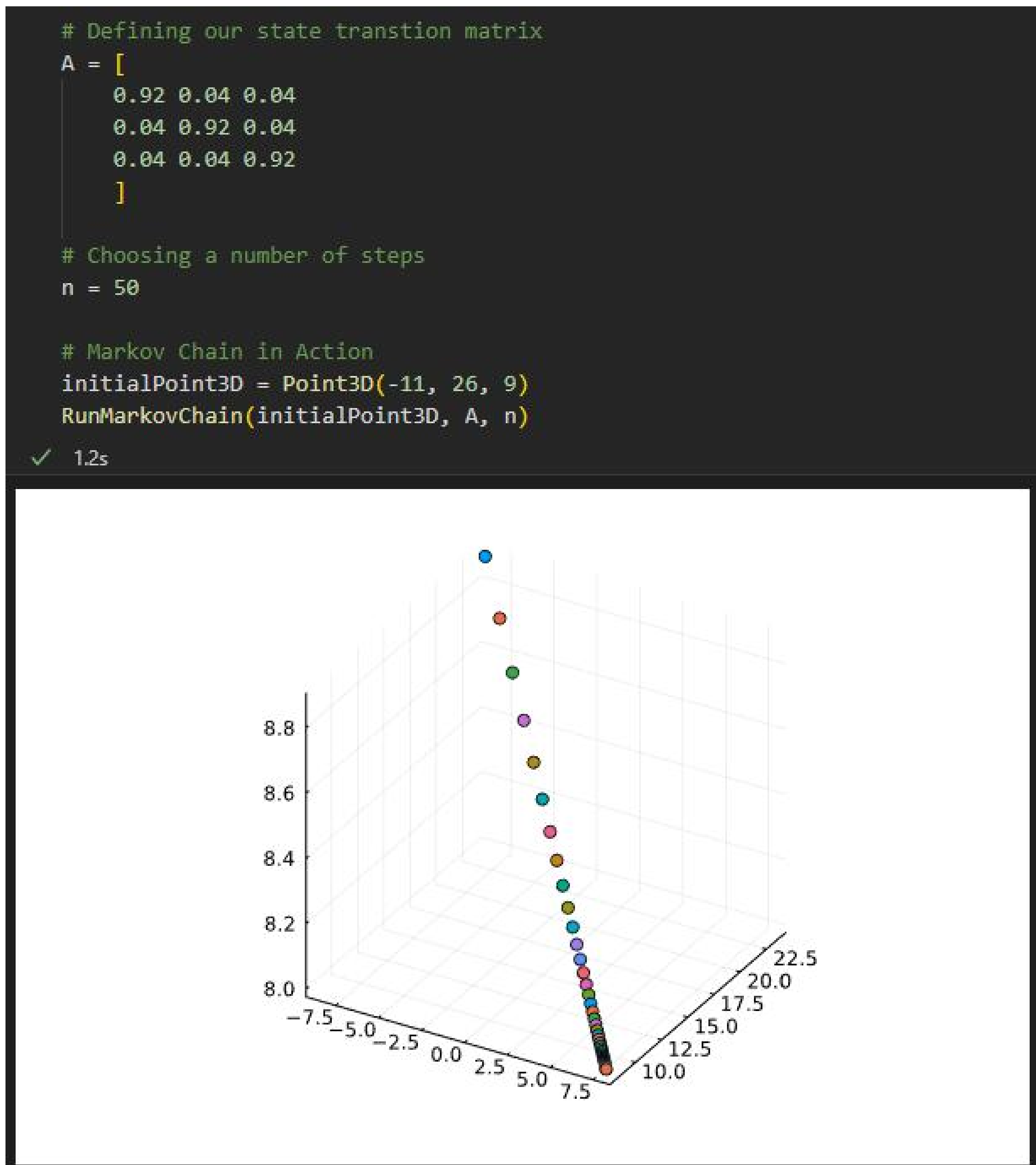
```
# Defining our state transtion matrix
A = [
    0.95 0.05
    0.05 0.95
    ]

# Choosing a number of steps
n = 50

# Markov Chain in Action
initialPoint2D = Point2D(-11, 26)
RunMarkovChain(initialPoint2D, A, n)
```

✓ 10.3s

Resultado da aplicação da cadeia de markov até a convergência

b) Pontos 3D

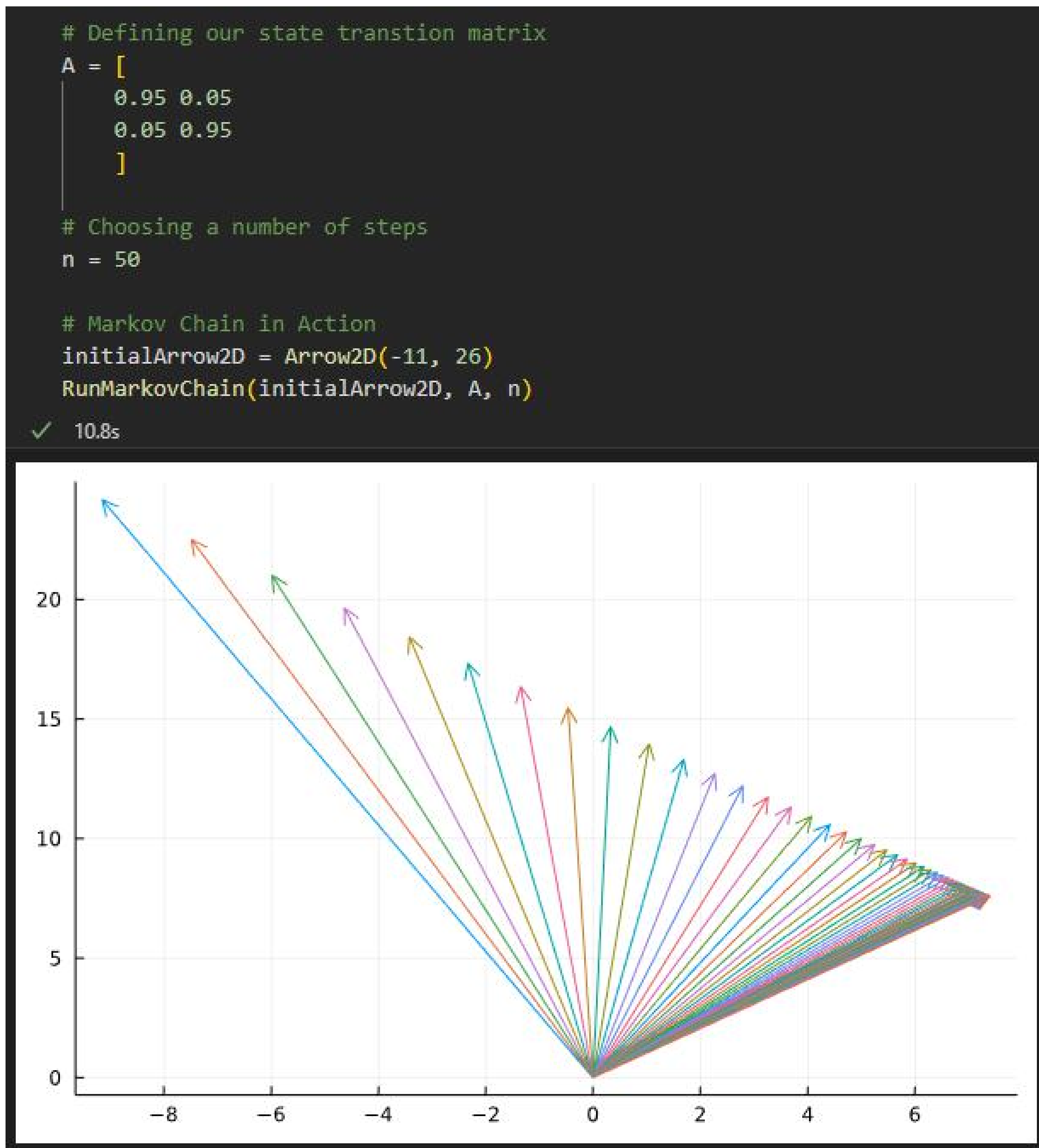
Figura 57 – Cadeia de Markov em Pontos 3D



Resultado da aplicação da cadeia de markov até a convergência

c) Setas 2D

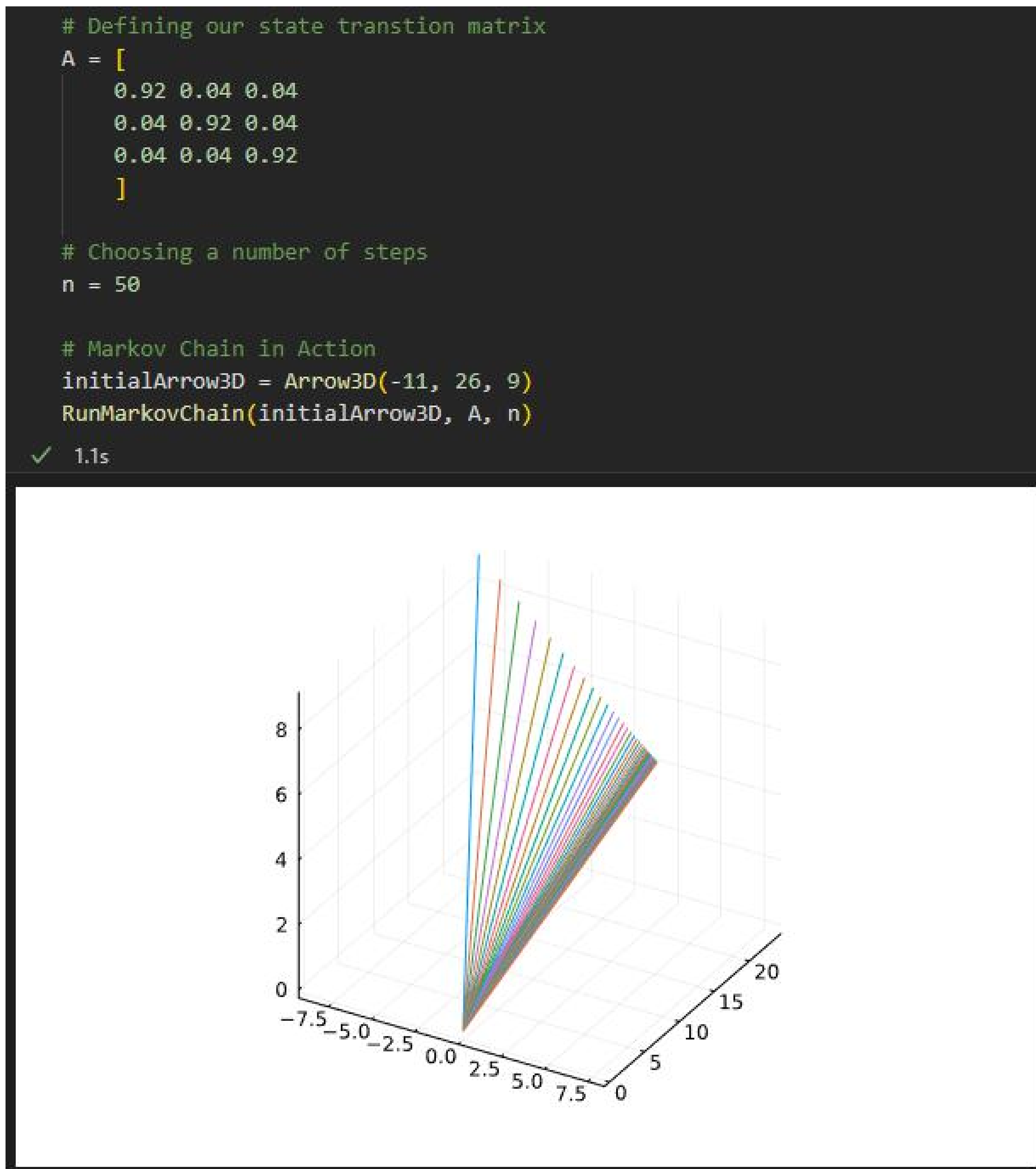
Figura 58 – Cadeia de Markov em Setas 2D



Resultado da aplicação da cadeia de markov até a convergência

d) Setas 3D

Figura 59 – Cadeia de Markov em Setas 3D



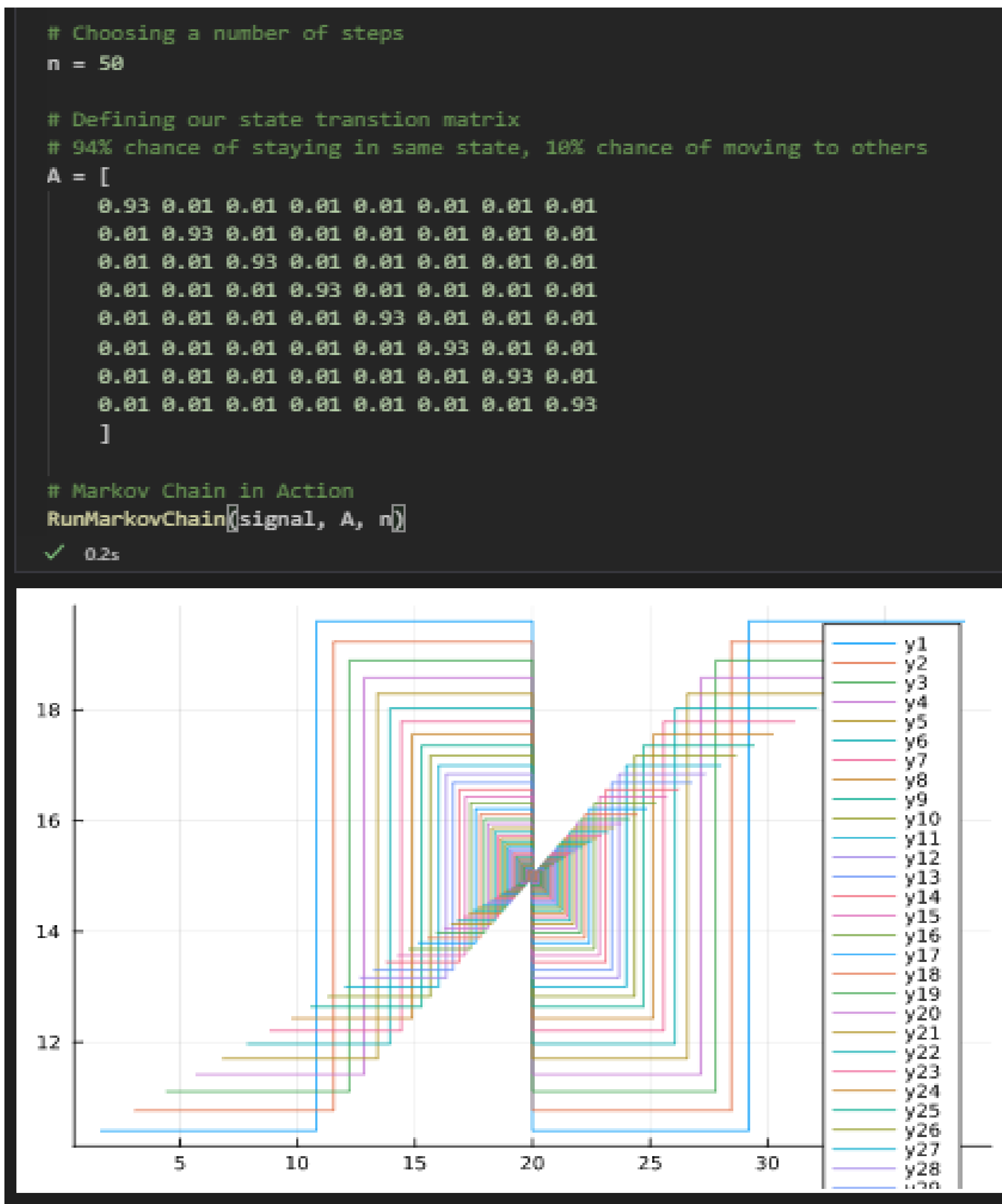
Resultado da aplicação da cadeia de markov até a convergência

e) Sinais

Figura 60 – Sinal inicial para exemplo de Cadeia de Markov

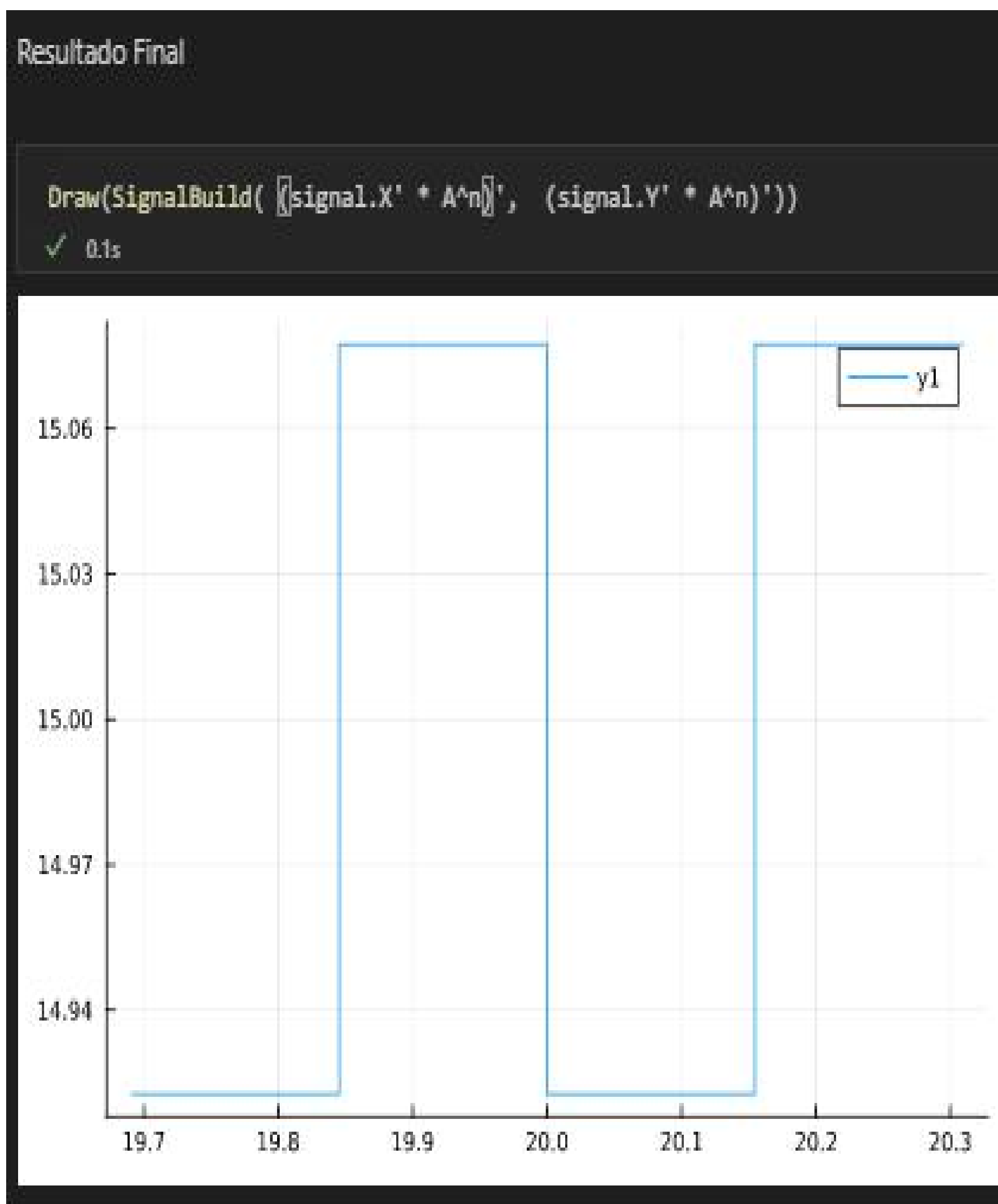
Sinal inicial

Figura 61 – Aplicação da Cadeia de Markov em sinal



Resultado da aplicação da cadeia de markov

Figura 62 – Sinal após convergência



Visualização após convergência

4 EASYLINALG PARA DESENVOLVEDORES

Neste capítulo será mostrado todo o desenvolvimento da biblioteca EasyLinalg, cobrindo todas as decisões e os aspectos que levaram à maneira que foi implementada. Como também será abordado neste capítulo, EasyLinalg é uma biblioteca de software livre e, sendo assim, as próximas sessões têm como fim servir de guia para a continuidade da biblioteca. É esperado que, ao final desse capítulo, o leitor entenda a estrutura de arquivos, bem como a comunicação entre os mesmos, os testes unitários, os tipos implementados, a forma de implementar novos tipos e quaisquer outros aspectos necessários para a continuidade da aplicação. Ao término da leitura deste capítulo, espera-se que um colaborador tenha ferramentas para efetuar a manutenção e desenvolvimento contínuo da biblioteca, dado que o universo de álgebra linear é extenso e tem-se espaço de sobra para novos recursos serem implementados.

4.1 MOTIVO DA ESCOLHA DA JULIA LANG

Antes de falar sobre qualquer outro aspecto, é necessário abordar a primeira decisão do projeto, a utilização de Julia como linguagem do projeto. A resposta mais simples para essa escolha é o fato da linguagem possuir sintaxe bem próxima à sintaxe da matemática, o que se adéqua quando o objetivo é simplificar demonstrações de álgebra linear. Esse não é o único aspecto dessa decisão, pois seguindo a mesma linha de raciocínio, o MATLAB tem essa mesma abordagem e segundo o Stack Overflow Survey 2021 (OVERFLOW, 2021) tem uma popularidade cerca de 3 vezes maior do que Julia.

O segundo grande fator para a escolha da linguagem vem do fato de Julia ser código aberto, sendo esse um fator que democratiza o acesso à ferramenta, tendo então um grande peso para a decisão da escolha da linguagem. Neste sentido, vale ressaltar que Python também foi uma das linguagens cotadas para o projeto, pois é uma linguagem amplamente utilizada acadêmica e profissionalmente, com uma ampla gama de ferramentas de álgebra linear. Python esteve próximo a ser a linguagem escolhida, dado o ponderamento de pontos positivos e negativos das opções que tínhamos em fase de prototipagem. Por fim, Julia foi escolhida, pois foi dado mais importância na simplificação que sua sintaxe pode gerar para o usuário final, sendo assim, estando mais alinhada com o objetivo do projeto.

4.2 DESIGN TÉCNICO

A partir da definição da linguagem, é possível iniciar então o design técnico da solução, assim escolhendo os padrões de engenharia de software que mais se adéquam ao objetivo.

Nas próximas subseções, serão mencionados alguns pontos de escolha técnica tomadas durante a prototipação e o desenvolvimento do projeto.

4.2.1 Motivadores

Seguindo a premissa de toda decisão técnica ser tomada embasada nos desafios que as motivaram, é necessário explicar os fatores associados às tomadas de decisões. Essa subseção planeja levantar alguns pontos que serão de extrema importância no desenvolver deste capítulo, para assim equalizar o conhecimento e permitir a observação dos problemas pelo mesmo ponto de vista dos desenvolvedores.

A primeira questão a ser abordada é um ponto previsto ao se usar Julia e o fato da linguagem ser multi-paradigma, ou seja, orientada a objetos e, em simultâneo, funcional (JULIA, s.d.a). Com isso, alguns recursos da orientação a objetos acabam sendo implementados de maneira diferente, assim, a fim de atingir o nível necessário de abstração, estes paradigmas tiveram de ser misturados. O que motiva então, nas próximas sessões, explanações de alguns conceitos básicos e como eles são aplicados na EasyLinalg.

Outro ponto importante é a continuidade e evolução da iniciativa. Sendo esse um projeto de final de curso, o escopo acaba sendo reduzido, então, para a evolução do projeto é ideal que exista uma colaboração futura, o que justificará alguns pontos de decisão abordados a seguir.

4.2.2 Código Aberto

De modo a promover a evolução da EasyLinalg, o foco de desenvolvimento sempre foi um código limpo, escalável e aberto. Ter o código aberto é a maneira mais viável de promover uma evolução saudável ao código, buscando criar uma comunidade em volta da solução para que sua evolução ocorra de maneira orgânica conforme a vontade desta comunidade. O projeto usa o modelo de desenvolvimento de software livre conhecido como Bazar (RAYMOND, 1999), onde o código é desenvolvido transparentemente e publicado na internet, sem omissão de nenhum trecho de código durante o desenvolvimento.

Foi utilizado esse modelo na biblioteca por acreditarmos que a maior abertura de todos os processos de desenvolvimento pode construir um produto mais sólido e consistente, além de promover o acesso à ferramenta. Este é o mesmo modelo adotado pelo desenvolvimento do Kernel Linux, o que demonstra que pode ser um modelo promissor.

Com isso é esperado que a biblioteca consiga seguir um rumo de desenvolvimento onde suas necessidades de evolução sejam ditadas pela vontade daqueles que utilizam e assim atinjam o objetivo de facilitar o ensino de álgebra linear.

A biblioteca atualmente pode ser acessada no *github* através do link:

`https://github.com/brunohry/EasyLinalg`

Por fim, a construção da biblioteca foi baseada na documentação de criação de pacotes de Julia (JULIA, s.d.b), de modo a proporcionar a facilitação no uso.

4.2.3 Tipos de Dados

Antes de contextualizar tipos à EasyLinalg, é preciso mencionar pontos de como a linguagem Julia implementa sua orientação a objetos. Um tipo de dado é uma estrutura que associa a ela propriedades e métodos, se aproximando do conceito clássico de uma classe na orientação a objetos. Seus atributos são carregados na definição do tipo, onde deve ser definido como uma estrutura de dados. Já os métodos são funções definidas fora do escopo do tipo, onde a associação entre função e tipo de dado ocorre na definição de tipos de dados dos parâmetros de entrada.

Um exemplo de implementação de um tipo, é o Zero lógico que mostramos abaixo. A implementação as seguir foi feita por Alan Edelman, um dos criadores da linguagem Julia, e é utilizada também na EasyLinalg:

Código 1 – Zero Lógico

```
struct Zero end   #zero Matrix


begin
    import Base: +,*,-

    Base.zero(::Type{Any})= Zero()
    +(::Zero, ::Zero) = Zero()
    -(::Zero, A) = -A
    +(::Zero, A) = A
    +(A,::Zero) = A
    *(::Zero, ::Zero) = Zero()
    *(X, ::Zero) = Zero()


end
```

Para a biblioteca, os tipos são utilizados como recursos para implementar os objetos da EasyLinalg. Criando tipos de dados especializados em cada objeto de estudo (como Seta, Ponto, Imagem e Sinal), pode-se desenvolver métodos específicos para aqueles objetos. Criando então um conjunto de métodos de mesmo nome para cada tipo, onde o comportamento de entrada e saída é o mesmo independente do tipo, cria-se a interface da EasyLinalg. Para a EasyLinalg, os tipos são construídos como estruturas que carregam dados e metadados referentes a uma interpretação de algum objeto de estudo. Então, os tipos implementados são essencialmente representações de objetos do instrumental educacional de álgebra linear.

Os tipos também carregam as redefinições dos sinais-base do Julia, como o operador de adição ou o operador de subtração. Essas redefinições são importantes, pois uma vez definidos novos tipos, as operações entre eles não estão definidas e devem ser descritas, para assim o comportamento das operações seguir o esperado na álgebra linear. Esses sinais ficam no pacote *Base* de Julia. Um exemplo disso é que podemos redefinir o operador de adição, "+", para somar dois objetos do tipo Sinal. Sendo que nesta definição é realizada uma verificação antes de somar efetivamente, conferindo então se os dois sinais estão distribuídos no mesmo espaço, ou seja, se estão distribuídos no mesmo intervalo do eixo x.

Uma ilustração da utilização de tipos na EasyLinalg são vetores, onde sua representação geométrica mais comum é uma seta em duas dimensões em um plano cartesiano. A estrutura de dados referente a esse tipo 2D na EasyLinalg pode ser carregada apenas tendo informações nos eixos x e y, ou seja, se deseja representar um vetor (seta em duas dimensões) cuja ponta está nas coordenadas $x = 1$ e $y = 2$, bastaria apenas fazer a construção desse tipo, como disposto a seguir, que automaticamente a outra extremidade da seta é considerada na origem $(0, 0)$:

```
Arrow2D( 1, 2 )
```

No caso do tipo chamado Sinal, os dados são basicamente referentes ao comportamento de uma função contínua. Um dos parâmetros é um intervalo no eixo x (segundo parâmetro do construtor) e o outro, uma função para o eixo y (primeiro parâmetro):

```
SignalBuildU( x -> sin(4.0 * x), -10.0:0.1:10.0 )
```

Neste caso será construído um sinal a partir da função $seno(4x)$, onde esse sinal será avaliado no intervalo x de -10 a 10, com a avaliação da função ocorrendo a cada intervalo de 0,1.

Todas as funções de construção dos tipos da EasyLinalg estão referenciadas na Tabela 1

4.2.4 Heranças

O paradigma da programação chamado de herança é um mecanismo que permite que objetos compartilhem atributos e métodos entre si. A utilização de tipos em Julia também permite a herança, facilitando assim que comportamentos sejam herdados de um tipo superior na hierarquia, gerando grande reaproveitamento de código.

Herança é utilizada na EasyLinalg de modo a facilitar o desenvolvimento dos tipos, sem a necessidade de implementar do zero todo o comportamento vetorial. Então seu principal uso na biblioteca é na definição dos tipos de dados da EasyLinalg. Os tipos de dados da biblioteca são implementados a partir da adição de códigos específicos da

representação daquele objeto de estudo em uma estrutura que herda o comportamento de *Abstract Array*. Com isso é possível construir um objeto de comportamento vetorial aproveitando toda a lógica deste comportamento já implementada em Julia, fazendo o foco do desenvolvimento ser a especialização que aquele tipo contém.

Os tipos da EasyLinalg são normalmente instâncias de *Abstract Vector*. Um *Abstract Vector* é um *Abstract Array* de apenas uma dimensão. No que lhe concerne, um *Abstract Array* (JULIA, s.d.a) é um tipo que herda um conjunto de comportamentos vetoriais, como, por exemplo, indexação multidimensional. Com os tipos da EasyLinalg herdando as características de um *Abstract Vector*, é possível reaproveitar todas as lógicas de vetores já implementadas em Julia. Para este comportamento funcionar é preciso definir algumas funções para tratamento de estruturas de dados (JULIA, s.d.a). A seguir exemplos das funções que devem ser definidas:

- size(). Esta função retorna as dimensões do objeto.
- getindex(i). Esta função retorna o valor alocado na posição i do objeto.
- setindex(v, i). Esta função aloca um valor v na posição i do objeto.

4.2.5 Polimorfismo

Polimorfismo é caracterizado por objetos diferentes que apresentam os mesmos métodos, ou seja, a mesma interface para interação com o objeto. Isso permite que uma mesma função seja chamada para dois objetos de naturezas completamente distintas e performe o resultado esperado, mesmo que para isso existam implementações diferentes de uma função com mesma assinatura.

Para aplicar o polimorfismo, utilizamos funções declaradas para os tipos construídos. Um exemplo desta utilização são as redefinições de size() para cada tipo, como no exemplo a seguir:

```
size(a::Arrow2D) = size([a.i,a.j])
size(x::Signal) = size(x.Y)
```

Para funções que podem ser reaproveitadas entre alguns tipos, o Julia permite a indicação de mais de um tipo nos parâmetros de entrada na declaração das funções. Isso é feito através da união e um exemplo dessa união ocorre entre os pontos 2D e 3D e setas 2D e 3D, tendo a sintaxe a seguir:

```
types = Union{Point2D, Point3D, Arrow2D, Arrow3D}
```

Esta união, por sua vez, será indicada como tipo dos parâmetros de entrada na declaração da função. Um exemplo de função que utiliza a união é o Draw(). O Draw() pode desenhar um vetor com vários objetos do mesmo tipo, ou apenas um único objeto.

Porem a implementação para desenhar um único objeto, seria a mesma de desenhar um vetor com somente 1 objeto do tipo. Então esta função é declarada apenas para desenhar o vetor de um tipo. O Draw() para um único objeto é implementado através de uma função envelope, que pega um dado e o coloca em um vetor, para assim chamar o Draw() específico para o vetor daquele tipo já implementado. Portanto, esta função envelope recebe qualquer tipo pertencente a uma união, não necessitando de reescrita de código para cada tipo. A seguir, a definição da função envelope, onde é chamada internamente o Draw() especializado para cada tipo:

```
function Draw(A::types, separate = false)
    return Draw([A], separate)
end
```

Com a construção de *Abstract Array* dos tipos, somado a essas definições de *union* e polimorfismo, muitas funções são construídas apenas uma vez e aplicadas a todos os tipos. O polimorfismo se mostra necessário em algumas exceções, onde uma função genérica pode não funcionar, podendo então ser definida a mesma função para cada tipo de entrada, ocultando assim a diferença de código através do polimorfismo. As funções de Draw no projeto são um grande exemplo desse polimorfismo, visto que se tem uma implementação para cada tipo, porém fica completamente invisível para o usuário que existem funções diferentes.

4.2.6 Árvore de arquivos

A árvore de arquivos foi criada de maneira que sua construção seja compatível com o empacotador de bibliotecas da Julia (JULIA, s.d.b). Mantendo essa compatibilidade, separamos a raiz em 3 diretórios principais:

a) docs
   - Responsável por armazenar toda documentação e exemplos de uso da biblioteca

b) src
   - Responsável por armazenar todo o código-fonte da biblioteca
   - O subdiretório basics contém um arquivo com as funções polimórficas bases da EasyLinalg
   - O subdiretório types, é dividido em subdiretórios para cada tipo de dado, os quais carregam todas as especializações dele
   - O arquivo EasyLinalg.jl é responsável por centralizar as importações de código dos tipos implementados para facilitar o uso da biblioteca

c) test
   - Responsável por armazenar os testes da biblioteca

- É dividida em subdiretórios para cada tipo, onde os testes de cada tipo estão nos seus respectivos diretórios. O arquivo runtests.jl é responsável por rodar os testes de todos os tipos.

Abaixo o esquema atual da árvore de arquivos da EasyLinalg.

```
/
├── docs
│   └── Exemplos
├── src
│   ├── basics
│   │   ├── draws.jl
│   │   └── operations.jl
│   ├── types
│   │   ├── arrow2D
│   │   │   └── arrow2DDefinitions.jl
│   │   ├── arrow3D
│   │   │   └── arrow3DDefinitions.jl
│   │   ├── image
│   │   │   └── imageDefinitions.jl
│   │   ├── ...
│   │   └── zero
│   │       └── zeroDefinitions.jl
│   └── EasyLinalg.jl
├── test
│   ├── arrow2D
│   │   └── arrow2DTeste.jl
│   ├── arrow3D
│   │   └── arrow3DTeste.jl
│   ├── image
│   │   └── imageTeste.jl
│   ├── ...
│   ├── zero
│   │   └── zeroTeste.jl
│   └── runtestes.jl
├── Manifes.toml
├── Project.toml
└── readme.md
```

Essa árvore de arquivos é distribuída de maneira a separar conceitos e facilita a manutenção e evolução do código, visto que é simples saber onde estão os códigos genéricos e compartilhados. Assim, um colaborador do código pode ter a segurança que uma alteração em um tipo não afetará outros e uma alteração numa função polimórfica afetará todos os tipos que usam da mesma. Assim, aquele que está contribuindo para o código consegue mensurar os impactos de suas modificações.

4.2.7 Testes unitários

Testes unitários têm como principal funcionalidade validar comportamentos no fluxo do código-fonte das menores unidades da aplicação. Utilizar testes unitários é uma boa tática para verificar se o desenvolvimento de uma nova funcionalidade está atingindo o resultado esperado, além de ser um tipo de teste que apresenta um custo relativamente baixo de desenvolvimento (SAWANT; BARI; CHAWAN, 2012). Esse teste visa localizar erros de comportamentos não previstos, por exemplo, dada uma função que depende de uma certa variável, se esta última for nula, o retorno da função pode não corresponder às expectativas. Aplicar testes unitários é uma boa prática de engenharia de software e reduz o número de bugs introduzidos no projeto.

Dado as múltiplas camadas da EasyLinalg, testes são essenciais para evitar a introdução de falhas na evolução do projeto. Com isso, utilizamos os utilitários de teste do Julia para realizar esses testes. Os testes são construídos para cada tipo da biblioteca, responsáveis por garantir que funcionalidades adicionadas estejam com o comportamento esperado, assim como assegurar que não foram introduzidas falhas na base de código atual. Para rodar os testes, é necessário ir até a pasta raiz do projeto e executar o comando:

```
Julia> ] test
```

Dividimos então esses testes em 3 sessões, são elas:

a) BaseOps
   - Responsável por implementar os testes das funções redefinidas do pacote Base do Julia. Um exemplo dos testes que devem ser desenvolvidos nela é se a operação de soma “+” para um novo tipo está apresentando o comportamento esperado

b) BuildOps
   - Responsável por implementar os testes das construções de cada um dos tipos

c) EasyLinalg
   - Responsável por testar se todas as funções declaradas na EasyLinalg estão com o comportamento esperado para o tipo em teste

4.3 IMPLEMENTAÇÃO

Finalizando a sessão de Design Técnico (4.2), esta sessão mostrará como foram efetivamente implementados todos os pontos abordados anteriormente.

4.3.1 Tipos definidos

Para começar a apresentar a implementação da EasyLinalg, é importante compreender os tipos implementados e suas construções para explorar as suas particularidades. A seguir será descrito cada tipo implementado junto ao código-fonte dos mesmos.

a) Ponto 2D

- Um ponto 2D herda o comportamento de um *AbstractVector* e tem sua construção limitada ao tamanho de duas coordenadas. Seu primeiro elemento representa a coordenada x e o segundo, a coordenada y no plano cartesiano.

  Ao redefinir as funções getindex() e setindex(), foram adicionadas validações de tamanho com erros personalizados no caso da utilização de tamanhos maiores que duas dimensões. Esses métodos são necessários para a linguagem saber acessar um índice em uma estrutura tanto para leitura, no caso do *get*, quando para a escrita, no caso do *setindex*.

  Para o ponto 2D são definidas as operações de soma e subtração entre dois pontos 2D, multiplicação por um escalar, divisão por um escalar.

  Por fim, a redefinição das funções de cópia também foram extremamente importantes, dado que um array sempre é propagado por referência e é importante que as funções de cópia funcionem e gerem objetos independentes na memória.
- Estrutura

  Código 2 – Estrutura ponto 2D

  ```
  struct Point2D <: AbstractVector{Number}
      x
      y
  end
  ```
- Redefinições

Código 3 – Redefinições ponto 2D

```
begin
    import Base: +,*,-,/,size,copy

    +(a::Point2D,b::Point2D) = Point2D(a.x+b.x, a.y+b.y)
    -(a::Point2D,b::Point2D) =  Point2D(a.x-b.x, a.y-b.y)

    *(b::Real,a::Point2D) = Point2D(a.x * b, a.y * b)
    *(a::Point2D,b::Real) = *(b::Real,a::Point2D)

    /(b::Real,a::Point2D) = Point2D(a.x / b, a.y / b)

    size(a::Point2D) = size([a.x,a.y])

    Base.copy(a::Point2D) = Point2D( deepcopy(a.x), deepcopy(a.y
       ) )
    Base.deepcopy(a::Point2D) = Point2D( deepcopy(a.x), deepcopy
       (a.y) )

    Base.getindex(a::Point2D, i::Int) = i > 2 || i < 1 ?
        error("Point2D can only have size 2") :
        [a.x,a.y][i]
    Base.setindex!(a::Point2D, v, i::Int) = i > 2 || i < 1 ?
        error("Point2D can only have size 2") :
        (i == 1 ? (a.x = v) : (a.y = v) )

end
```

b) Ponto 3D

- Um ponto 3D tem seu funcionamento completamente análogo ao de um ponto 2D, apenas tendo uma dimensão a mais, correspondente a dimensão z no plano cartesiano. A seguir, sua estrutura exemplifica a adição de uma coordenada.
- Estrutura

Código 4 – Estrutura ponto 3D

```
mutable struct Point3D <: AbstractVector{Number}
    x
    y
    z
end
```

c) Setas 2D

- A construção de uma seta 2D é análoga ao código-fonte de um ponto 2D, suas diferenças se dão apenas nas funções de saída, visto que uma seta 2D é centrada na origem, suas diferenças para um ponto 2D viram apenas sintáticas.

– Estrutura

Código 5 – Estrutura seta 2D

```
mutable struct Arrow2D <: AbstractVector{Number}
    i
    j
end
```

d) Seta 3D

– Uma seta 3D tem seu funcionamento completamente análogo ao de uma seta 2D, apenas tendo uma dimensão a mais, correspondente a coordenada z no plano cartesiano.

– Estrutura

Código 6 – Estrutura seta 3D

```
mutable struct Arrow3D <: AbstractVector{Number}
    i
    j
    k
end
```

e) Sinal

– Descrição

Um sinal é uma representação de uma função contínua avaliada em um intervalo. Sua implementação é uma herança de um *AbstractVector* e sua construção tem duas variáveis, são elas:

X é um vetor de valores do eixo x. Seus dados são considerados metadados do tipo. Sendo assim, uma vez definido o vetor X, não pode ser modificado após a construção do objeto.

Y é um vetor de valores do eixo y. Seus dados são considerados os dados principais do tipo, e são eles que serão acessados através das funções getindex() e setindex().

X e Y têm a mesma quantidade de elementos e cada posição i avaliada em X deve ser diretamente ligada à posição i avaliada em Y, formando assim um ponto. Então (X[i], Y[i]) é um ponto válido no intervalo de observação para todo i no tamanho de X e de Y.

Para o sinal são definidas as operações de soma e subtração entre dois sinais desde que os dois tenham a variável X exatamente igual, multiplicação por um escalar, divisão por um escalar, produto interno entre dois sinais de variáveis X exatamente iguais.

– Estrutura

Código 7 – Estrutura sinal

```
struct Signal <: AbstractVector{Number}
    X
    Y
end
```

Particularmente, um sinal tem dois construtores, são eles:

- `SignalBuildU(fx, x )`

  O qual constrói um sinal a partir de uma função f(x) e uma avaliação dela no intervalo dado (x)

Código 8 – SignalBuildU

```
function SignalBuildU(fx, x )
    return Signal(collect(x), fx.(x))
end
```

- `SignalBuild(x, y )`

  O qual constrói um sinal a partir de 2 vetores, x e y, que devem ser de tamanhos iguais. O vetor x será mapeado diretamente para X do sinal e o vetor y será mapeado diretamente para Y do sinal.

Código 9 – SignalBuild

```
function SignalBuild(x, y )
    if(size(x) != size(y))
        error("To Build a Signal, the size of x and y must be
           the same")
    end
    return Signal(x, y)
end
```

Por fim, a redefinição das funções de cópia também é necessária dado que um array sempre é propagado por referência. É importante que as funções de cópia funcionem e gerem objetos independentes na memória.

- Redefinições

Código 10 – Redefinindo sinal

```
function dotProduct(x::Signal,y::Signal)
    sum = 0
    for i=1:size(x.Y)[1]
        sum += x.Y[i] * y.Y[i]
    end
    return sum
end

begin
    import Base: +,*,-,/,size,copy,vect
    # addition rule
    +(x::Signal,y::Signal) = y.X == x.X ? Signal(x.X, x.Y + y.Y)
        : error("X array must be equal for every entry")
    -(x::Signal,y::Signal) = y.X == x.X ? Signal(x.X, x.Y - y.Y)
        : error("X array must be equal for every entry")

    # dot product
    *(x::Signal,y::Signal) = y.X == x.X ? dotProduct(x,y) :
        error("X array must be equal for every entry")

    # multiplying by scalar
    *(y::Real,x::Signal) = Signal(x.X, x.Y * y)
    *(x::Signal,y::Real) = *(y::Real,x::Signal)

    /(y::Real,x::Signal) = Signal(x.X, x.Y / y)

    # adicionando safecheck na expressÃčo [A, B]
    function Base.vect(K::Signal...)
        for i=1:length(K)
            j = i+1
            if j <= length(K) && K[i].X != K[j].X
                return error("X array must be equal for every
                    entry")
            end
        end
        return copyto!(Vector{Signal}(undef, length(K)), K)
    end

    size(x::Signal) = size(x.Y)

    Base.copy(x::Signal) = Signal(copy(x.X), copy(x.Y))

    Base.getindex(x::Signal, i::Int) = Base.getindex(x.Y,i)

    Base.setindex!(x::Signal, v, i::Int) = (x.Y[i] = v)

end
```

f) Imagem

– Descrição

Uma imagem é uma representação em forma de matriz onde cada ponto da imagem tem seu valor como uma representação de uma cor no padrão "RGB". Imagine uma imagem de 1920 $\times$ 1080 píxels de resolução, agora pense que cada píxel é uma coordenada no plano cartesiano. Diretamente é possível associar cada píxel a uma coordenada para estruturar uma matriz que representaria cada imagem, mas ainda não teríamos uma boa visualização da imagem caso tivéssemos uma representação binária para cada entrada da matriz.

Para que a imagem tome forma, cor e seja interpretável aos nossos olhos, o computador precisa saber como lidar com cada informação das entradas da matriz, portanto cada valor da matriz pode ser a representação de cor no RGB, mencionada anteriormente, para que a linguagem imprima na tela algo que seja identificável por nós, humanos.

– Estrutura

A estrutura desse tipo é extremamente simples, mantendo a facilidade para o usuário fazer a construção do tipo, só precisando ter o arquivo em um diretório acessível pelo código, tal construção se dá através da chamada:

```
Image(‘‘caminho\ate\arquivo.jpg’’)
```

Código 11 – Estrutura imagem

```
using Images

struct Image <:AbstractVector{Vector{RGB}}
    I
end
```

– Redefinições

A biblioteca aceita a construção com o parâmetro do construtor sendo uma string com o caminho do arquivo e também o parâmetro sendo uma matriz de "Float64" sendo cada float o valor do RGB.

Código 12 – Redefinindo imagem

```
begin
    import Base: +,*,-,^,/,size,zero,copy,vect

    function Image(Name::String)
        img = load(Name)
        vect = Vector{Vector{RGB}}()
        for i in 1:size(img, 2)
           push!(vect, img[:,i] )
        end
        return Image(vect)
    end

    function Image(img::Matrix{RGB{Float64}})
        vect = Vector{Vector{RGB}}()
        for i in 1:size(img, 2)
           push!(vect, img[:,i] )
        end
        return Image(vect)
    end

    function ToNativeImage(img::Image)
        return reduce(hcat,img.I)
    end

    # addition rule
    +(x::Image,y::Image) = Image(x.I + y.I)
    -(x::Image,y::Image) =  Image(x.I - y.I)

    # multiplying by scalar
    *(y::Number,x::Image) = Image(x.I * y)
    *(x::Image,y::Real) = *(y::Number,x::Image)

    /(y::Number,x::Image) = Image(x.I / y)


    size(x::Image) = size(x.I)

    Base.copy(x::Image) = Image(deepcopy(x.I))
    Base.deepcopy(x::Image) = Image(deepcopy(x.I))

    Base.getindex(x::Image, I::Int) = x.I[I]

    Base.setindex!(x::Image, v, I::Int, J::Int) = (x.I[I, j] = v
       )
    Base.convert(::Type{Image}, a::Matrix{RGB{T}}) where{T} =
       Image(a)

    Base.zero(::Type{Vector{RGB}}) = Zero()
end
```

4.3.2 Draws

A saída de execuções da biblioteca é toda centralizada nas funções Draw(). Essa é uma categoria de função de característica polimórfica e cada tipo da biblioteca tem sua própria implementação, dado que o tipo é o que define a forma de exibição do dado. Existe basicamente uma implementação de Draw() para cada categoria de dado da EasyLinalg, como ponto 2D, sinal, imagem.

As funções Draw() tem finalidade de imprimir na tela gráficos referentes a visualização de resultados ou tipos passados no parâmetro. A função basicamente recebe um array de tipo e retorna na tela a visualização do array recebido. O usuário pode então realizar operações e manipulações a sua vontade e visualizar em um gráfico 2D ou 3D, dependendo dos tipos operados, como ficou o resultado. A criação dessa função torna a biblioteca visual e facilita que o usuário tenha mais proximidade com as manipulações que estão sendo feitas, uma vez que saímos exclusivamente do campo de linguagem de programação e vamos para as interpretações gráficas.

A declaração do método é a mesma para quaisquer tipos da biblioteca, o que muda é sua implementação. Logo, ao chamar a função, o usuário não precisa se preocupar em configurar uma saída para aquele tipo. Em outras palavras, o usuário só precisa chamar a função e passar o objeto que deseja ver desenhado. Todas as funções Draw() são implementadas para array dos tipos pertencentes à EasyLinalg e, por padrão, vão desenhar todos os itens do array sobrepostos. Para que os plots de cada item do array aconteçam separados, deve-se utilizar o parâmetro $separate = true$ na chamada da função.

A seguir, veremos um exemplo da construção de Draw(). No começo do código 13 vemos o que caracteriza o comportamento polimórfico do Draw(), dado pela função envelope citada na sessão 4.2.5, e então vamos à implementação de Draw() para vetor de Arrow2D.

Código 13 – draws.jl

```
types = Union{Signal, Point2D, Point3D, Arrow2D, Arrow3D}

function Draw(A::types, clear = true, separate = false)
    return Draw([A], clear, separate)
end


function Draw(A::Vector{Arrow2D}, clear = true, separate = false)
    if(clear)
        plot()
    end
    m=size(A)[1]
    plotsX = []
    plotsY = []
    for i=1:m
        append!(plotsX, [[0, A[i].i]])
        append!(plotsY, [[0, A[i].j]])
    end

    if(!separate)
        p = plot!(plotsX[1], plotsY[1], arrow=true, label="")
        for i=2:size(plotsY)[1]
            plot!(p, plotsX[i], plotsY[i], arrow=true, label="")
        end
        return p
    end

    return plot!(plotsX,plotsY, arrow=true, layout = (m, 1), label="")
end
```

4.3.3 Operações internas

Operações internas são funções implementadas que não necessitam ser especializadas para cada tipo. Dentro desta seção estão implementações genéricas, como as funções toNumberVector() e toNumberMatrix(), que traduzem um tipo da EasyLinalg para um vetor numérico ou matriz numérica, respectivamente.

Código 14 – operations-example.jl

```
types = Union{Signal, Point2D, Point3D, Arrow2D, Arrow3D}

function toNumberMatrix(S::Vector{<:types})
    K = zeros(size(S[1])[1], size(S)[1])
    for i=1:size(S)[1]
        K[:,i] = toNumberVector(S[i])
    end
    return K
end

function toNumberVector(S::types)
    K = zeros(size(S)[1])
    for j=1:size(S)[1]
        K[j] = getindex(S, j)
    end
    return K
end
```

A utilização de toNumberVector() e toNumberMatrix() se dá ao realizar algumas operações onde o tipo é convertido para vetor ou matriz numérica, operado e então convertido novamente ao tipo. Um exemplo dessa utilização pode ser visto no Código 15, que representa a multiplicação entre um vetor de um tipo da EasyLinalg e uma matriz numérica. Por fim, o Código 15 também é um dos exemplos de Operações internas.

Código 15 – product.jl

```
function *( S::Vector{<:types}, w::Matrix{<:numberTypes})
    # Buscando tipo dos elementos de S
    element = deepcopy(S[1])
    type = typeof(element)

    # Criando Vetor de saida no mesmo tipo do vetor original
    K = Vector{type}([element])

    # Tranformando vetor original em matriz numÃľrica
    T = toNumberMatrix(S)

    # Realizando operaaao *
    T = T * w

    # Convertendo resultado para o tipo de S
    s = size(T)[2]
    for i=1:s
        for j=1:size(T)[1]
            K[i][j] = T[j,i]
        end
        if( i < s)
            push!(K, deepcopy(element))
        end
    end

    # Retornando K no mesmo tipo de S
    return K
end
```

4.3.4 Testes Unitários

Testes unitários visam manter o correto comportamento das funcionalidades implementadas na biblioteca, aumentando a confiabilidade dos desenvolvedores nas modificações realizadas, uma vez que um teste unitário valida se o comportamento dos fluxos do código estão seguindo o rumo desejado. Imagine o exemplo de soma entre duas setas 2D, o resultado deveria ser também uma seta 2D com os valores devidamente somados, logo o teste unitário para esse caso validaria se após a soma, o resultado seria uma seta 2D com os valores certos.

Os testes unitários são implementados para cada tipo novo da EasyLinalg, portanto idealmente um desenvolvedor da biblioteca deve criar testes unitários para cada novo tipo ou operações criados, garantindo que não hajam fluxos quebrados ou com resultados inesperados.

A seguir, temos um exemplo de implementação de teste unitário para Seta 2D, de

modo a ilustrar a implementação dos testes.

Código 16 – arrow2DTeste.jl

```
@testset verbose = true "Arrow2D Testes" begin
    @testset "Build Ops" begin
        # test construction function
        # test type of concatenation
        @test typeof([Arrow2D(1,2), Arrow2D(3,4)]) == Vector{Arrow2D}
    end;
    @testset "Base Ops" begin
        @test Arrow2D(1,2) + Arrow2D(3,4) == Arrow2D(4,6)
        @test Arrow2D(4,6) - Arrow2D(3,4) == Arrow2D(1,2)
        @test Arrow2D(1,2) * 2 == Arrow2D(2,4)
        @test 2 * Arrow2D(1,2) == Arrow2D(2,4)
        @test Arrow2D(2,4) / 2 == Arrow2D(1,2)
    end;
    @testset "Safe Checks" begin
        @test_throws MethodError Arrow2D(1.0, 2.0, 3.0)
        @test_throws ErrorException Arrow2D(1.0, 2.0)[3]
        @test_throws ErrorException Arrow2D(1.0, 2.0)[3] = 3.0
    end;
    @testset "EasyLinalg" begin
        @test convex_combination( [ Arrow2D(1.0, 2.0), Arrow2D(4.0, 8.0)
            ], 0.5) == [ Arrow2D(1.0, 2.0), Arrow2D(2.5, 5), Arrow2D
            (4.0, 8.0) ]
        @test toNumberVector(Arrow2D(1.0, 2.0)) == [1.0, 2.0]
        @test toNumberMatrix( [ Arrow2D(1.0, 2.0), Arrow2D(4.0, 5.0) ] )
            == [1.0 4.0; 2.0 5.0]
        @test ([ Arrow2D(1.0, 2.0), Arrow2D(4.0, 5.0) ]) * [ 1.0 0.0;
            0.0 1.0] ==  [ Arrow2D(1.0, 2.0), Arrow2D(4.0, 5.0) ]
        @test ([ Arrow2D(1.0, 2.0), Arrow2D(4.0, 5.0) ]) * [ 2.0, 3.0 ]
            ==  Arrow2D(14.0, 19.0)
        @test ([ Arrow2D(1.0, 2.0), Arrow2D( 4.0, 5.0) ]) * [ 2.0 3.0;
            4.0 5.0 ] == [ Arrow2D(18.0, 24.0), Arrow2D( 23.0, 31.0 ) ]
        @test ([ Arrow2D(1.0, 2.0), Arrow2D( 4.0, 5.0) ]) \ (([ Arrow2D
            (1.0, 2.0), Arrow2D( 4.0, 5.0) ]) * [ 2.0 3.0; 4.0 5.0 ]) ==
            [ 2.0 3.0; 4.0 5.0 ]
        @test ([ Arrow2D(1.0, 2.0), Arrow2D( 4.0, 5.0), Arrow2D(7.0,
            8.0) ]) \ (([ Arrow2D(1.0, 2.0), Arrow2D( 4.0, 5.0), Arrow2D(
             7.0, 8.0) ]) * [ 2.0 3.0; 4.0 5.0; 6.0 7.0]) âĽĹ [ 2.0 3.0;
            4.0 5.0; 6.0 7.0]
    end;
end;
```

O arquivo runtests.jl concatena todos os testes da biblioteca para assim rodar todos os testes em sequência. Sendo assim, para executar todos os testes da biblioteca, basta executar o comando:

```
julia ./runtests.jl
```

O resultado dos testes tem a seguinte saída esperada:

Figura 63 – Resultado dos testes unitários

```
Test Summary:  | Pass  Total
Arrow2D Testes |   17     17
  Build Ops    |    1      1
  Base Ops     |    5      5
  Safe Checks  |    3      3
  EasyLinalg   |    8      8
```

Resultado da execução do teste de Setas 2D

5 CONSIDERAÇÕES FINAIS

5.1 CONCLUSÃO

Este trabalho final propôs o desenvolvimento de uma biblioteca que possibilita a construção de exemplos da disciplina de álgebra linear. Após o devido planejamento e implementação, foi possível construir exemplos de maneira simples conforme ilustrando na sessão 3.4, o que caracteriza que o resultado foi atingido. Entretanto, um ponto de interesse seria levar a biblioteca para dentro de salas de aulas reais e notar a percepção dos alunos e professores. Devido a nuances referentes a prazos e o próprio escopo de mínimo produto viável da solução, não possível fazer a observação da biblioteca em uma sala de aula. Esse experimento poderia ser mais uma validação da proposta do projeto, pois se teria o parecer de pessoas que realmente utilizaram a ferramenta pelas primeiras vezes, podendo então relatar suas impressões, suas dores e sugestões. A experiência de trazer a biblioteca para uma sala de aula é sugerida como um trabalho futuro na seção 5.2.

Um dos objetivos do trabalho era construir um projeto com foco em facilitar sua utilização e evolução. Para isso, este texto visa ser documentação do que foi implementado, sua construção foi planejada para que ao término da leitura, tanto pessoas da área da matemática quanto da computação pudessem entender e usufruir da biblioteca assumindo os papéis de usuário ou colaborador. Já, para a realização técnica de tal objetivo, foram necessárias horas de estudo para a devida arquitetação da solução. Assim foi gerada uma biblioteca de Julia que segue a documentação e padrões da linguagem, facilitando a novos colaboradores ajudar na evolução da mesma.

Para os usuários foi proposta uma maneira simplificada a interação entre usuário e biblioteca, sendo então necessário poucos comandos para se utilizar todas as funcionalidades implementadas. Através dessa simplificação é proporcionada autonomia ao usuário, assim o facilitando a explorar os exemplos utilizados em aula e modificar os mesmos. Alterando parâmetros de entrada, usando outros tipos e criando seus próprios exemplos, é entregue para o usuário a possibilidade de usar sua criatividade para entender melhor no que consiste a matéria estudada.

Um dos grandes conceitos-chave do trabalho foi a criação de uma álgebra linear onde seus objetos são baseados em tipos. Isso possibilita a amarração entre o valor semântico daquele objeto ao próprio objeto. Através de tipos então é possível implementar toda iteração daquele objeto com os conceitos da álgebra linear sem repassar esse trabalho para o usuário. Novos tipos podem então ser criados na biblioteca, para assim conseguir gerar um portfólio cada vez mais amplo de exemplos, como demonstrado no capítulo 4. Por fim, ao se construir esta biblioteca, teve-se extensa preocupação em generalizar o que estava sendo desenvolvido, para que futuros desenvolvedores pudessem se aproveitar das

construções já existentes, assim desenvolvendo um código sem muitas particularizações.

5.2 EVOLUÇÃO E TRABALHOS FUTUROS

Entendemos que a EasyLinalg ainda não é uma solução completa para o ensino de álgebra linear, mesmo que no começo do curso de Ciência da Computação na Universidade Federal do Rio de Janeiro. Sabendo que ainda existe a necessidade de evolução na biblioteca, o que é entregue através desse trabalho é o mínimo produto viável da solução, esperando então que a biblioteca possa evoluir conforme a necessidade dos usuários. Algumas pessoas podem vislumbrar diferentes formas de explorar o que fora entregue com a biblioteca, enquanto outras podem já esquematizar como montar modelos para salas de aula. Como a solução possui código-fonte aberto, qualquer pessoa pode usufruir como desejar, seja somente utilizando as funcionalidades existentes ou implementando novas. Devido às suas experiências, professores podem sentir falta de ferramentas que atendam seus modelos de aula, podendo então eles terem ideias e implementar novas funcionalidade. Em contrapartida, do outro lado do ensino-aprendizado dos professores, temos os alunos, que normalmente estudam os conteúdos da disciplina pela primeira vez, podem ter vontade de se aventurar desenvolvendo pequenos trechos de código com o que aprenderam em uma aula.

Os direcionamentos futuros desse trabalho final podem envolver o teste e a utilização do projeto em instituições de ensino, não se prendendo somente a salas de aulas físicas, mas também em cursos online, mentorias ou *bootcamps*. Outra possibilidade seria o desenvolvimento de um sistema com interface gráfica, como um site, aplicação *desktop* ou *mobile* onde o usuário possa ter uma experiência mais desprendida do código, tanto na visualização quanto na construção dos exemplos. O estado onde a interface gráfica fosse alcançada abrangeria mais pessoas do que fora visionado pelo escopo inicial do projeto.

REFERÊNCIAS

AGGARWAL, C. C.; AGGARWAL, L.-F.; LAGERSTROM-FIFE. **Linear algebra and optimization for machine learning**. [S.l.]: Springer, 2020. v. 156.

AYTEKIN, C.; KIYMAZ, Y. Teaching linear algebra supported by geogebra visualization environment. **Acta Didactica Napocensia**, ERIC, v. 12, n. 2, p. 75–96, 2019.

BEAUBOUEF, T. Why computer science students need math. **ACM SIGCSE Bulletin**, ACM New York, NY, USA, v. 34, n. 4, p. 57–59, 2002.

CALAFIORE, G. C.; GHAOUI, L. E. **Optimization models**. [S.l.]: Cambridge university press, 2014.

CAMPBELL-KELLY, M. Origin of computing. **Scientific American**, JSTOR, v. 301, n. 3, p. 62–69, 2009.

CORNELLI, G. et al. Fintech and big tech credit: a new database. BIS working paper, 2020.

GOTHELF, J. Using proto-personas for executive alignment. **UXMAG**, 2012. Disponível em: https://uxmag.com/articles/using-proto-personas-for-executive-alignment. Acesso em: 21 mai.2022.

JORGE, T. V.; COSTA, E. C. D. Análise das modalidades de contratações clt e pj para os profissionais de tecnologia da informação. **Revista Interface Tecnológica**, v. 18, n. 2, p. 91–104, 2021.

JULIA. **Julia 1.7 Documentation**. JULIA, s.d. Disponível em: https://docs.julialang.org/en/v1/. Acesso em: 21 mai.2022.

JULIA. **Pkg.jl Documentation**. JULIA, s.d. Disponível em: https://pkgdocs.julialang.org/v1/. Acesso em: 21 mai.2022.

JUPYTER. Jupyter project documentation. s.d. Disponível em: https://docs.jupyter.org/en/latest/. Acesso em: 21 mai.2022.

LIMA, P. V. P. de et al. Brasil no pisa (2003-2018): reflexões no campo da matemática. **TANGRAM-Revista de Educação Matemática**, v. 3, n. 2, p. 03–26, 2020.

LOPEZ, I. F.; SEGADAS, C. A disciplina cálculo i nos cursos de engenharia da ufrj: sua relação com o acesso à universidade e sua importância para a conclusão do curso. **Revista de Engenharia da Universidade Católica de Petrópolis**, v. 8, n. 2, p. 92–107, 2014.

MASOLA, W. de J.; ALLEVATO, N. Dificuldades de aprendizagem matemática de alunos ingressantes na educação superior. **Revista Brasileira de Ensino Superior**, 2016.

OVERFLOW, S. **Stack Overflow Developer Suvery 2021**. STACK OVERFLOW, 2021. Disponível em: https://insights.stackoverflow.com/survey/2021. Acesso em: 21 mai.2022.

RAYMOND, E. The cathedral and the bazaar. **Knowledge, Technology & Policy**, Springer, v. 12, n. 3, p. 23–49, 1999.

SAWANT, A. A.; BARI, P. H.; CHAWAN, P. Software testing techniques and strategies. **International Journal of Engineering Research and Applications (IJERA)**, v. 2, n. 3, p. 980–986, 2012.

SHIRLEY, P.; ASHIKHMIN, M.; MARSCHNER, S. **Fundamentals of computer graphics**. [S.l.]: AK Peters/CRC Press, 2009.

SIMÕES, M. de F.; FERRÃO, M. E. Competência percebida e desempenho escolar em matemática. **Estudos em Avaliação Educacional**, v. 16, n. 32, p. 25–42, 2005.

STRANG, G. **Linear algebra and its applications.** [S.l.]: Belmont, CA: Thomson, Brooks/Cole, 2006.

VAROQUAUX, G. et al. Scikit-learn: Machine learning without learning the machinery. **GetMobile: Mobile Computing and Communications**, ACM New York, NY, USA, v. 19, n. 1, p. 29–33, 2015.