UNIVERSIDADE FEDERAL DO CARIRI
CENTRO DE CIÊNCIAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM MATEMÁTICA EM REDE NACIONAL

WILIA ALBERTO DE SOUSA

O USO DO MANIM PARA O ENSINO DE MATEMÁTICA DO ENSINO MÉDIO

JUAZEIRO DO NORTE
2025

WILIA ALBERTO DE SOUSA

O USO DO MANIM PARA O ENSINO DE MATEMÁTICA DO ENSINO MÉDIO

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Matemática em Rede Nacional do Centro de Ciências e Tecnologia da Universidade Federal do Cariri, como parte dos requisitos necessários à obtenção do título de Mestre em Matemática. Área de concentração: Matemática na Educação Básica.

Orientador: Prof. Dr. Vicente Helano Feitosa Batista Sobrinho

JUAZEIRO DO NORTE
2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Cariri
Sistema de Bibliotecas

S725u Sousa, Wilia Alberto de.
O uso do Manim para o ensino de matemática do ensino médio / Wilia Alberto de Sousa. – 2025.
114 p. (Inclui bibliografia, p. 98–100).

Dissertação (Mestrado) – Universidade Federal do Cariri, Programa de Pós-graduação em Matemática em Rede Nacional (PROFMAT), Juazeiro do Norte, 2025.

Orientador: Prof. Dr. Vicente Helano Feitosa Batista Sobrinho

1. Animação digital. 2. Python. 3. Manim. I. Batista Sobrinho, Vicente Helano Feitosa – orientador. II. Título.

CDD 510

Bibliotecário: João Bosco Dumont do Nascimento – CRB 3/1355

WILIA ALBERTO DE SOUSA

O USO DO MANIM PARA O ENSINO DE MATEMÁTICA DO ENSINO MÉDIO

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Matemática em Rede Nacional do Centro de Ciências e Tecnologia da Universidade Federal do Cariri, como parte dos requisitos necessários à obtenção do título de Mestre em Matemática. Área de concentração: Matemática na Educação Básica.

Aprovada em: 28 de agosto de 2025.

BANCA EXAMINADORA

gov.br Documento assinado digitalmente
VICENTE HELANO FEITOSA BATISTA SOBRINHO
Data: 30/09/2025 14:49:13-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Vicente Helano Feitosa Batista Sobrinho
UFCA

gov.br Documento assinado digitalmente
VALDINES LEITE DE SOUSA JUNIOR
Data: 30/09/2025 13:02:35-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Valdinês Leite de Sousa Júnior
UFCA

gov.br Documento assinado digitalmente
RAFAEL PERAZZO BARBOSA MOTA
Data: 30/09/2025 09:21:24-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Rafael Perazzo Barbosa Mota
UFRPE

Dedico este trabalho àquela que sempre acreditou em mim, desde a infância, que representa o verdadeiro significado do amor e será eternamente minha maior inspiração: minha mãe.

Agradecimentos

Primeiramente, agradeço a Deus pelo dom da vida, pela força e pela perseverança que me sustentaram, mesmo diante dos inúmeros desafios surgidos ao longo desta caminhada.

À minha mãe, Maria Socorro de Sousa Alberto, pelo apoio constante e pelas palavras de incentivo em todos os momentos.

Aos meus irmãos, Daniel Alberto de Souza, Danilo Alberto de Souza, Romerito Alberto de Souza e os demais irmãos, por estarem ao meu lado e apoiarem minha trajetória acadêmica.

Aos professores do PROFMAT-UFCA, Dra. Érika Boizan, Dr. Francisco Pereira Chaves, Dr. Francisco de Assis, Dr. Valdinês Leite, Dr. Vicente Helano e Dr. Valdir Ferreira, pelo conhecimento compartilhado e pelo compromisso com a formação de seus alunos.

Ao meus orientador, Dr. Vicente Helano Feitosa Batista Sobrinho, pela orientação cuidadosa, disponibilidade e apoio constante.

Aos colegas de turma pela união e companheirismo durante toda a caminhada.

Aos colegas de trabalho, pela força que me deram nesse período de estudo.

Por fim, à Sociedade Brasileira de Matemática — SBM, pela oportunidade de cursar o PROFMAT em Rede Nacional, contribuindo de forma decisiva para minha formação.

Este trabalho foi realizado com o apoio financeiro da Coordenação de Aperfeiçoamento Pessoal de Ensino Superior (CAPES), no âmbito do Programa de Mestrado Profissional em Matemática em Rede Nacional (Profmat) da Universidade Federal do Cariri.

RESUMO

Esta pesquisa explora a biblioteca Manim como recurso didático no ensino de matemática. O trabalho desenvolvido inclui uma contextualização sobre o papel das animações digitais na educação, destacando as orientações da BNCC e documentos correlatos que evidenciam a importância do uso crítico e criativo de recursos computacionais na aprendizagem. São abordados, de forma introdutória, os fundamentos da linguagem Python e a estrutura da biblioteca Manim, enfatizando seu potencial para a criação de animações matemáticas. Também é discutida a relevância da integração entre pensamento computacional e ensino de matemática, especialmente na visualização e resolução de questões de avaliações externas, como o SAEPE. O produto desta dissertação é um conjunto de animações, organizadas em um repositório online, que possibilita que professores as apliquem e/ou adaptem os códigos para uso em sala de aula, produzindo materiais visuais e dinâmicos, favorecendo a compreensão conceitual e a participação ativa dos estudantes no processo de aprendizagem.

Palavras-chave: Animação digital. Python. Manim.

ABSTRACT

This research explores the Manim library as a didactic resource in mathematical teaching. The work developed comprises a contextualization on the role of computer animations in math education, based on the guidelines of the BNCC and related documents that emphasize the importance of the critical and creative use of computational resources in learning. The fundamentals of Python programming and the Manim library are introduced, emphasizing their potential for creating mathematical animations. The relevance of integrating computational thinking into mathematical teaching is also discussed, especially with regard to visualization and resolution of problems from external assessments, such as SAEPE. The product of this dissertation is a set of animations, organized in an online repository, enabling teachers to apply and/or adapt their codes for classroom use, thus producing visual and dynamic materials that foster conceptual understanding and active student participation in the learning process.

Keywords: Computer animation. Python. Manim.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

EJA	Educação de Jovens e Adultos, p. 5
Enem	Exame Nacional do Ensino Médio, p. 4
Ideb	Índice de Desenvolvimento da Educação Básica, p. 4
PISA	Programme for International Student Assessment, p. 4
SAEPE	Sistema de Avaliação da Educação de Pernambuco, p. 4
SEDUC-CE	Secretaria da Educação do Estado, p. 5
SEE-PE	Secretaria de Educação do Estado de Pernambuco, p. 5
SPAECE	Sistema Permanente de Avaliação da Educação Básica do Ceará, p. 4
Saeb	Sistema de Avaliação da Educação Básica, p. 4
TIMSS	Trends in International Mathematics and Science Study, p. 4

Sumário

Capítulo 1

Introdução

A Matemática, por ser uma ciência exata, exige um raciocínio bastante abstrato, especialmente no que diz respeito à compreensão e manipulação dos conteúdos, como funções, trigonometria, geometria analítica, entre outros. Essa característica faz com que muitos alunos desenvolvam uma certa resistência à disciplina, frequentemente vista e rotulada como a mais difícil. No entanto, na era da informação em que vivemos, onde a disseminação de conhecimento ocorre de modo contínuo e rápido, abordagens diferenciadas aliadas à tecnologia tem impulsionado mudanças significativas no ensino da matemática para torná-la mais acessível.

O uso de ferramentas digitais vem ganhando cada vez mais espaço nas práticas pedagógicas voltadas ao ensino da matemática. Essas tecnologias oferecem a professores e alunos uma nova perspectiva para abordagem da disciplina. Ao visualizar e manipular representações gráficas, por exemplo, é possível perceber as relações entre diferentes elementos matemáticos de maneira mais clara, o que facilita a compreensão e a fixação do conteúdo. Nesse contexto, Lima e Rocha (2022) ressalta que o uso das tecnologias digitais pode contribuir para uma melhor compreensão da matemática em assuntos como geometria, álgebra e aritmética, ao apresentar aplicações que fazem sentido para os educandos.

Os softwares educacionais, no contexto do ensino da matemática, desempenham um papel bastante importante para as práticas pedagógicas atualmente. Ferramentas como GeoGebra e Desmos permitem que alunos e professores explorem conceitos matemáticos de forma dinâmica, promovendo uma aprendizagem mais visual e interativa. Esses recursos favorecem a experimentação e a construção do conhecimento, aproximando o estudante de uma abordagem investigativa da matemática. Borba e Penteado (2016) destacam que o uso das tecnologias digitais não apenas amplia as possibilidades metodológicas, mas também influencia a maneira como os alunos desenvolvem o pensamento matemático, proporcionando novas formas de representação e interpretação dos conceitos.

Além dos softwares, as plataformas digitais voltadas para o ensino da matemática, como Khan Academy (2025) e PhET (2025), têm se consolidado como ferramentas que auxiliam na personalização da aprendizagem. Essas plataformas oferecem um ambiente em que os alunos podem avançar no próprio ritmo e revisar conteúdos conforme suas necessidades, favorecendo um aprendizado mais autônomo e significativo. Valente, Freire e Arantes (2018) ressalta que a tecnologia educacional deve ser integrada ao ensino de maneira a estimular o pensamento crítico e a participação ativa dos estudantes, permitindo que construam o conhecimento matemático por meio da experimentação e da resolução de problemas.

Ferramentas como Microsoft Excel, Google Sheets e LibreOffice Calc podem se tornar grandes aliadas no âmbito educacional, especialmente quando usadas para trabalhar com dados de forma prática. Além disso, são amplamente utilizadas em diversos trabalhos organizacionais, o que torna seu uso em sala de aula ainda mais relevante. Com elas, os alunos podem criar tabelas, gerar gráficos interativos e aplicar fórmulas para resolver problemas de estatística, álgebra e modelagem matemática. Segundo Moran (2013), a inserção das tecnologias no ensino deve ir além da simples transmissão de conteúdos, possibilitando a exploração ativa dos conceitos e o desenvolvimento do pensamento crítico. Assim, ao aprender a usar planilhas para organizar orçamentos, analisar tendências ou calcular projeções, os alunos não apenas compreendem melhor os conteúdos matemáticos, mas também desenvolvem habilidades essenciais para o cotidiano e para suas futuras carreiras.

Nesse contexto, os vídeos têm se tornado uma ferramenta popular entre os jovens para aprendizado, especialmente com a ascensão de plataformas como o YouTube, que oferece acesso a uma enorme quantidade de conteúdos tutoriais e explicativos. Pesquisa realizada pela Pearson (2018) indica que muitos jovens preferem vídeos curtos e tutoriais como forma de estudar, já que permitem uma aprendizagem visual e possibilitam rever os conteúdos conforme necessário. Além disso, o uso de vídeos tem sido estudado e dado como um importante aliado, segundo levantamento da FUNIBER (2019), por sua capacidade de se conectar com os estudantes de uma maneira mais envolvente, já que eles podem, muitas vezes, ver pessoas da sua faixa etária explicando conceitos de forma mais acessível.

Dentre as várias tecnologias citadas acima para o ensino da matemática, as animações em vídeo têm se mostrado ferramentas promissoras, especialmente no contexto digital. Embora mais comuns na educação a distância, elas são destaque também no ensino presencial, à medida que recursos tecnológicos são cada vez mais incorporados ao ambiente escolar. As animações atuam como material didático, apresentando conteúdos por meio de imagens, sons, textos, efeitos e citações, o que envolve diferentes formas de linguagem (SILVA; FELCHER; FOLMER, 2024). Pesquisas realizadas por Silva (2011) e Rocato (2009) indicam que as animações

podem ser um recurso valioso no ensino da matemática, estimulando a imaginação, a visualização e a abstração dos alunos, o que facilita a compreensão de conceitos matemáticos desde os mais simples até os mais complexos.

Os resultados do Brasil em avaliações educacionais, como o PISA 2022, evidenciam desafios significativos no ensino de matemática. Atualmente, o país ocupa a 65ª posição em Matemática, 52ª em Leitura e 62ª em Ciências entre os 81 participantes, com pontuações médias de 379, 410 e 403 pontos, respectivamente, conforme matéria publicada no g1 (2023). Esses índices refletem a necessidade de revisar as abordagens pedagógicas adotadas, incorporando métodos mais interativos e que favoreçam o entendimento dos alunos. De acordo com Nóvoa (1992), a adaptação das práticas pedagógicas às necessidades dos alunos e o uso de recursos tecnológicos são fundamentais para melhorar os resultados educacionais. Ele enfatiza a importância de um ensino que vá além da transmissão de conteúdo, buscando formas mais dinâmicas e eficazes de engajamento dos estudantes, tornando o aprendizado mais significativo e aplicável à realidade do aluno.

Apesar da crescente disponibilidade de recursos audiovisuais educacionais, há carências de materiais que exploram o conteúdo matemático e a resolução de questões de maneira mais interativa. Nesse contexto, a biblioteca Manim, com sua capacidade de criar animações de alta qualidade, pode transformar a forma como os alunos aprendem e se preparam para exames como SAEPE, SAEB e ENEM, permitindo que visualizem a teoria e a resolução de problemas passo a passo e compreendam os conceitos de forma mais eficaz.

1.1 Sistemas de avaliação de ensino

Nas últimas décadas, a avaliação educacional consolidou-se como um dos pilares fundamentais para a formulação e implementação de políticas públicas no Brasil. A busca por indicadores precisos sobre a qualidade do ensino impulsionou a criação de um macrossistema de avaliação que abrange diferentes níveis e modalidades de ensino. Esse movimento resultou na estruturação de um conjunto abrangente de exames e índices que permitem não apenas mensurar o desempenho dos estudantes, mas também subsidiar a tomada de decisões estratégicas para a melhoria contínua da educação.

A experiência internacional e a realidade brasileira demonstram que as estratégias mais eficazes para aprimorar a qualidade do ensino são aquelas centradas nos resultados obtidos dessas avaliações, como aponta o estudo "O uso dos resultados das avaliações de aprendizagem no Brasil" (VELASCO, 2022). Nesse contexto, as avaliações em larga escala têm se consolidado como ferramentas essenciais para compreender a dinâmica dos sistemas educacionais, permitindo uma análise detalhada de

seus processos e resultados. Países com diferentes culturas e orientações ideológicas de governo têm adotado amplamente essas avaliações, evidenciando a importância desse mecanismo na formulação de políticas educacionais. Atualmente, 19 países da América Latina contam com sistemas nacionais de avaliação educacional, e a participação nessas iniciativas tem se expandido para avaliações internacionais de grande relevância, como o *Programme for International Student Assessment* (PISA) e o *Trends in International Mathematics and Science Study* (TIMSS). Essas avaliações, conduzidas em parceria com nações da União Europeia, América do Norte, Ásia e África, reforçam a tendência global de mensuração da qualidade educacional com base em parâmetros comuns.

Atualmente, o Brasil conta com uma diversidade de instrumentos avaliativos, entre os quais se destacam:

- Exame Nacional do Ensino Médio (Enem)
- Sistema de Avaliação da Educação Básica (Saeb)
- Sistema Permanente de Avaliação da Educação Básica do Ceará (SPAECE)
- Sistema de Avaliação da Educação de Pernambuco (SAEPE)

Enem

Instituído em 1998, o Enem tem como objetivo avaliar o desempenho dos estudantes ao final da educação básica e servir como mecanismo de acesso ao ensino superior. Ele é composto por uma redação e quatro provas objetivas, cada uma com 45 questões, abrangendo as áreas: Linguagens, Códigos e suas Tecnologias; Ciências Humanas e suas Tecnologias; Ciências da Natureza e suas Tecnologias; Matemática e suas Tecnologias. Os resultados do Enem são utilizados como critério de acesso ao ensino superior em instituições públicas e privadas, além de servirem para a obtenção de bolsas de estudo e financiamentos estudantis.

Saeb

Criado em 1990, o Saeb tem a finalidade de analisar o nível de aprendizado dos alunos ao longo da educação básica e contribuir para o desenvolvimento de políticas educacionais. A avaliação é composta por testes aplicados a estudantes de diferentes etapas escolares, abordando conteúdos de Língua Portuguesa e Matemática, com foco na compreensão leitora e na resolução de problemas. Os dados obtidos pelo Saeb são utilizados para calcular o Índice de Desenvolvimento da Educação Básica (Ideb), servindo como referência para o acompanhamento do desempenho escolar e a implementação de melhorias no sistema de ensino.

SPAECE

O SPAECE foi estabelecido em 1992 pela Secretaria da Educação do Estado (SEDUC-CE) com o objetivo de monitorar e aprimorar a qualidade do ensino. Inicialmente voltado para os estudantes do ensino fundamental, sua abrangência foi expandida para incluir também o ensino médio e a Educação de Jovens e Adultos (EJA). A avaliação, realizada em larga escala, contempla as disciplinas de Língua Portuguesa e Matemática, além de investigar o contexto socioeconômico dos estudantes e seus hábitos de estudo. Questionários específicos também são aplicados a professores e gestores escolares, permitindo uma análise mais ampla do cenário educacional. Os resultados obtidos servem como base para a formulação e o aperfeiçoamento de políticas públicas voltadas à melhoria contínua do ensino.

SAEPE

O SAEPE foi implementado em 2000 pela Secretaria de Educação do Estado de Pernambuco (SEE-PE) com a finalidade de monitorar e aprimorar a qualidade do ensino na rede pública. Inicialmente destinado a estudantes do ensino fundamental, sua abrangência foi ampliada para incluir também o ensino médio. A avaliação, realizada em larga escala, abrange as disciplinas de Língua Portuguesa e Matemática para estudantes dos 3^{o}, 5^{o} e 9^{o} anos do Ensino Fundamental e do 3^{o} ano do Ensino Médio/Normal Médio. Os resultados acerca dessa avaliação são utilizados para subsidiar a formulação, implementação e monitoramento de políticas públicas educacionais. Além disso, servem como ferramenta para identificar áreas que necessitam de intervenções pedagógicas, orientar a alocação de recursos e promover o desenvolvimento de estratégias que visem a melhoria contínua do ensino.

Sua matriz de referência é baseada no modelo do SAEB, sendo organizada em temas e seus descritores. Para o 3^{o} ano do Ensino Médio, temos o seguinte:

1. GEOMETRIA
    - D01 – Identificar figuras semelhantes mediante o reconhecimento de relações de proporcionalidade.
    - D02 – Reconhecer aplicações das relações métricas do triângulo retângulo em um problema que envolva figuras planas ou espaciais.
    - D03 – Relacionar diferentes poliedros ou corpos redondos com suas planificações ou vistas.
    - D04 – Identificar a relação entre o número de vértices, faces e/ou arestas de poliedros expressa em um problema.
    - D05 – Resolver problema que envolva razões trigonométricas no triângulo retângulo (seno, cosseno, tangente).

- D06 – Identificar a localização de pontos no plano cartesiano.
- D07 – Interpretar geometricamente os coeficientes da equação de uma reta.
- D08 – Identificar a equação de uma reta apresentada a partir de dois pontos dados ou de um ponto e sua inclinação.
- D09 – Relacionar a determinação do ponto de interseção de duas ou mais retas com a resolução de um sistema de equações com duas incógnitas.
- D10 – Reconhecer, dentre as equações do 2º grau com duas incógnitas, as que representam circunferências.

2. GRANDEZAS E MEDIDAS

- D11 – Resolver problema envolvendo perímetro de figuras planas.
- D12 – Resolver problema envolvendo área de figuras planas.
- D13 – Resolver problema envolvendo a área total e/ou volume de um sólido (prisma, pirâmide, cilindro, cone, esfera).

3. NÚMEROS E OPERAÇÕES/ÁLGEBRA E FUNÇÕES

- D14 – Identificar a localização de números reais na reta numérica.
- D15 – Resolver problema que envolva variação proporcional, direta ou inversa, entre grandezas.
- D16 – Resolver problema que envolva porcentagem.
- D17 – Resolver problema envolvendo equação do 2º grau.
- D18 – Reconhecer expressão algébrica que representa uma função a partir de uma tabela.
- D19 – Resolver problema envolvendo uma função do 1º grau.
- D20 – Analisar crescimento/decrescimento, zeros de funções reais apresentadas em gráficos.
- D21 – Resolver problema envolvendo PA/PG dada a fórmula do termo geral.
- D22 – Reconhecer o gráfico de uma função polinomial de 1º grau por meio de seus coeficientes.
- D23 – Reconhecer a representação algébrica de uma função do 1º grau dado o seu gráfico ou vice-versa.
- D24 – Resolver problemas que envolvam os pontos de máximo ou de mínimo de uma função polinomial do 2º grau.

- D25 – Relacionar as raízes de um polinômio com sua decomposição em fatores do 1º grau.
- D26 – Identificar a representação algébrica e/ou gráfica de uma função exponencial.
- D27 – Identificar a representação algébrica e/ou gráfica de uma função logarítmica, reconhecendo-a como inversa da função exponencial.
- D28 – Resolver problema que envolva função exponencial.
- D29 – Identificar gráficos de funções trigonométricas (seno, cosseno, tangente) reconhecendo suas propriedades.
- D30 – Determinar a solução de um sistema linear.
- D35 – Identificar o gráfico que representa uma situação descrita em um texto.

4. ESTATÍSTICA, PROBABILIDADE E COMBINATÓRIA

- D31 – Resolver problema de contagem utilizando o princípio multiplicativo ou noções de permutação simples, arranjo simples e/ou combinação simples.
- D32 – Resolver problema que envolva probabilidade de um evento.
- D33 – Resolver problema envolvendo informações apresentadas em tabelas e/ou gráficos.
- D34 – Associar informações apresentadas em listas e/ou tabelas simples aos gráficos que as representam e vice-versa.

Além dessa matriz, em que consta os conteúdos que o discente deve conhecer ao término do ensino médio, são definidos padrões de desempenhos, usados para avaliar com precisão a eficácia do processo educacional. Esses padrões permitem identificar as áreas em que os estudantes demonstram domínio, bem como aquelas que necessitam de intervenções pedagógicas. Na área de matemática, os padrões de desempenho do SAEPE estão estruturados da seguinte forma:

- ELEMENTAR I (DE 0 ATÉ 250 PONTOS)

  Neste nível, os alunos demonstram habilidades básicas em matemática, como reconhecer formas geométricas simples, realizar operações aritméticas básicas com números inteiros e decimais, interpretar gráficos e tabelas simples, e resolver problemas simples de proporcionalidade e porcentagem.

- ELEMENTAR II (251 A 290 PONTOS)

Os alunos deste nível apresentam habilidades matemáticas fundamentais, como reconhecer formas geométricas mais complexas, realizar operações aritméticas com números inteiros e decimais, incluindo frações, resolver problemas de proporcionalidade e porcentagem mais complexos, e interpretar gráficos e tabelas mais complexos.

- BÁSICO (291 A 325 PONTOS)

  Os discentes que atingem esse nível demonstram um entendimento mais profundo de conceitos matemáticos e são capazes de aplicar suas habilidades em situações mais complexas. Eles podem resolver problemas de geometria envolvendo áreas e volumes, realizar operações com números racionais, resolver equações e inequações simples, e interpretar gráficos e tabelas com mais de duas variáveis.

- DESEJÁVEL (326 PONTOS A MAIS)

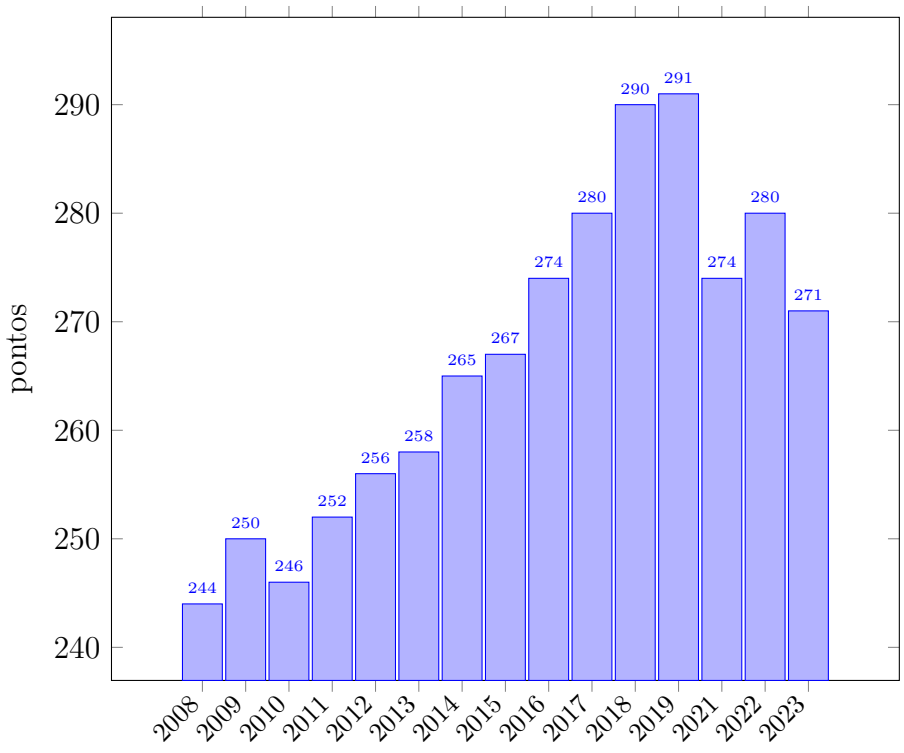
  Nesta etapa, os alunos apresentam domínio sólido de conceitos matemáticos e são capazes de aplicar suas habilidades em situações complexas e abstratas. Eles podem resolver problemas de geometria envolvendo trigonometria e geometria analítica, realizar operações com números reais, resolver equações e inequações complexas, interpretar gráficos e tabelas com múltiplas variáveis e realizar cálculos de probabilidade e estatística.

Segundo o SAEPE, para uma escola ser considerada eficaz, ela deve proporcionar padrões de aprendizagem adequados a todos os estudantes, independentemente de suas características individuais, familiares e sociais. Se apenas um grupo de estudantes consegue aprender com suficiente qualidade o que é ensinado, aumentam as desigualdades educacionais e, como consequência, elevam-se os indicadores de repetência, evasão e abandono escolar.

Embora o Estado de Pernambuco venha apresentando pequenos avanços no desempenho dos alunos do Ensino Médio, os resultados se concentram nos níveis mais baixos da escala de proficiência. A Figura 1.1 contém as pontuações médias obtidas no SAEPE pelos alunos do 3º ano do Ensino Médio da rede estadual, no período compreendido entre os anos de 2008 a 2023. Os dados apresentados foram extraídos do Power Retri, ferramenta que mapeia o desempenho escolar dos estudantes da rede estadual, elaborada pelo governo do estado.

Em 2008, a proficiência média em Matemática era de 244 pontos, classificada no padrão de desempenho Elementar I, o mais baixo da escala. Nos anos seguintes, houve um leve crescimento, ultrapassando a barreira dos 250 pontos em 2011, o que resultou na mudança para a categoria Elementar II. Esse avanço, contudo, não foi

Figura 1.1: Pontuação média no SAEPE de Matemática dos estudantes da rede estadual do 3º Ano do Ensino Médio no período de 2008 a 2023.



Fonte: Elaborado pelo autor.

suficiente para alcançar níveis mais elevados de proficiência, se mantendo dentro do mesmo padrão ao longo da década seguinte.

Entre 2016 e 2019, a proficiência atingiu seus maiores índices, chegando a 290 pontos em 2018. Entretanto, a partir de 2021, observa-se um declínio considerável nos resultados, com a pontuação caindo para 274, mantendo-se nesse patamar até 2023. Esses resultados evidenciam havia uma tendência crescente do desempenho dos estudantes pernambucanos até ocorrer uma queda abrupta ocasionada pela pandemia de COVID-19.

1.2 Animações no ensino de matemática

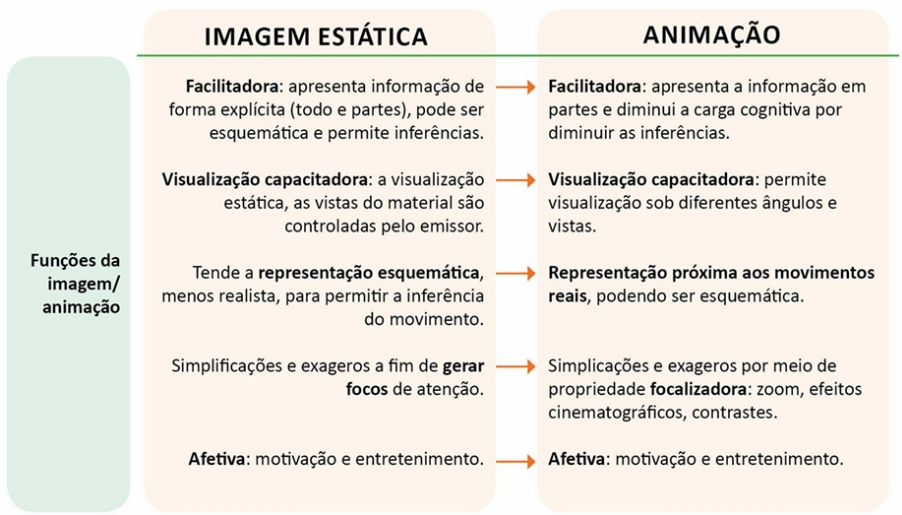
De maneira geral, o uso de animações em contextos educacionais remonta ao início da década de 1980, sendo empregadas para diversas finalidades, como despertar o interesse dos alunos por meio de personagens atrativos, representar conceitos abstratos ou ilustrar fenômenos que não são facilmente observáveis no cotidiano. Além disso, elas permitem a visualização de elementos que não possuem uma forma concreta, como a execução de algoritmos computacionais (AINSWORTH, 2008). Embora a utilização de recursos gráficos com propósitos didáticos não seja uma novidade, visto

que representações visuais, como desenhos, acompanham a história da educação, as animações oferecem um potencial ainda maior. Segundo Alves, Battaiola e Spinillo (2014), esses recursos ampliam a capacidade de compreensão ao proporcionar representações dinâmicas, tornando o aprendizado mais acessível e envolvente.

As motivações para o uso de animações no ensino variam conforme a necessidade pedagógica, mas, em geral, estão associadas à percepção de que esses recursos são particularmente envolventes e, portanto, capazes de favorecer o aprendizado. Além disso, as animações auxiliam na compreensão de conceitos complexos e atendem a demandas cognitivas inerentes ao processo de aprendizagem. No entanto, há aqueles que encaram seu uso com ceticismo, argumentando que seu emprego deve ser moderado e criterioso (AINSWORTH, 2008).

A Figura 1.2 contém uma comparação das funções de imagens estáticas e animações segundo Alves, Battaiola e Spinillo (2014).

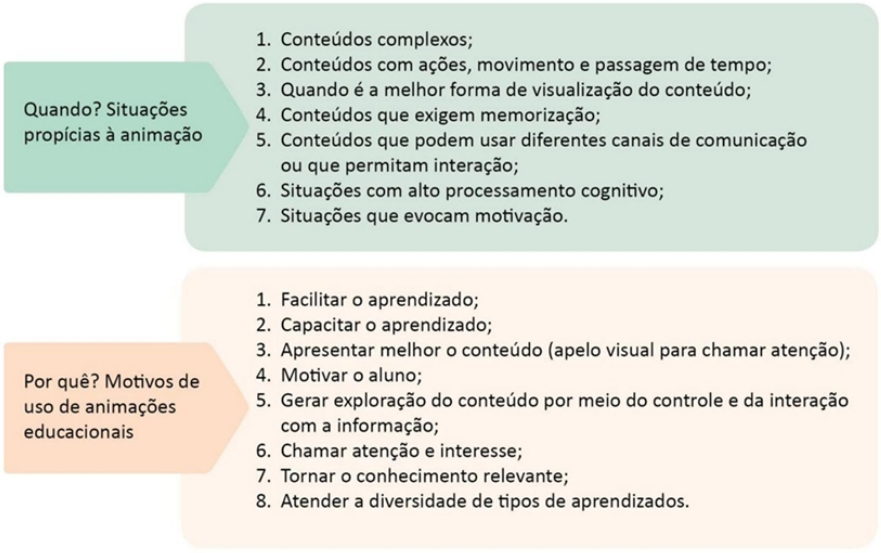
Figura 1.2: Comparação das funções de imagens estáticas e animações.



Fonte: Alves, Battaiola e Spinillo (2014)

É fundamental adotar uma abordagem criteriosa ao transformar um conteúdo em animação, garantindo que essa escolha seja avaliada e devidamente justificada. Isso porque a elaboração da animação não deve ser encarado apenas como uma mera alteração na forma de representação do conteúdo, mas sim como um processo de estruturação mais organizado e coerente, sustentado por um estudo prévio e por uma necessidade específica. O objetivo é oferecer uma nova perspectiva das informações, potencializando sua compreensão e significado. Com essa premissa, Alves, Battaiola e Spinillo (2014) desenvolveram o esquema da Figura 1.3.

Figura 1.3: Quando e Porque animar?



Fonte: Alves, Battaiola e Spinillo (2014)

Além do que foi exposto, Ainsworth (2008) também faz referência à teoria cognitiva de Richard Mayer, considerada uma das mais completas sobre o uso de animações na aprendizagem. A teoria se baseia em três principais teses:

1. Existem dois canais distintos para processar representações visuais e verbais;
2. Cada canal tem capacidade limitada para processar informações;
3. A aprendizagem significativa dos alunos ocorre quando há seleção, organização e integração do novo material com o conhecimento prévio, contribuindo para a construção do entendimento.

Com base nisso, Mayer e Moreno (2002) concluiu que "os alunos aprendem de forma mais profunda quando combinam animação com narração, em vez de utilizar apenas narração", pois essa combinação ajuda a superar as limitações do sistema cognitivo.

Outro ponto destacado por Ainsworth (2008) é que as animações podem facilitar o processamento cognitivo de duas formas. A primeira, chamada função habilitante, diz respeito à capacidade das animações de representar elementos que não podem ser exibidos por imagens estáticas. A segunda, conhecida como função facilitadora, refere-se à ajuda que as animações oferecem aos alunos para criar modelos mentais dinâmicos de diferentes situações. Além disso, as animações podem ser úteis para reduzir conflitos cognitivos, já que sua reprodução visual pode ser mais realista do que uma simples descrição escrita ou falada, facilitando a compreensão e o alcance de resultados corretos pelos alunos.

Nota-se, então, que a utilização de animações não deve ocorrer de forma aleatória, mas sim quando há uma real necessidade de intervenção por meio desse recurso. Nesse sentido, Alves, Battaiola e Spinillo (2014) destacam que uma questão central nesta reflexão é ressaltar que a animação produzida não deve apenas apresentar o conteúdo, mas também cumprir seu papel como objeto de aprendizagem, atuando em outros aspectos do processo, como, por exemplo, questões motivacionais.

Para que essas animações possam ser criadas, há diversas tecnologias disponíveis, variando em complexidade, acessibilidade e propósito. Ferramentas tradicionais, como softwares de apresentação e reprodutores/editores de vídeo, permitem a elaboração de animações básicas por meio de transições, efeitos visuais e manipulação de imagens estáticas. A facilidade de uso dessas ferramentas contribui para sua ampla aceitação, tornando-as uma escolha popular entre educadores que buscam dinamizar o ensino sem a necessidade de conhecimentos técnicos avançados. Além disso, sua compatibilidade com diferentes dispositivos e formatos facilita a integração das animações em diversos contextos educacionais.

No campo da matemática, diversas iniciativas vêm sendo desenvolvidas para tornar o ensino e a aprendizagem mais acessíveis e dinâmicos. Entre estas, destaca-se o GeoGebra (2025), um software interativo que integra conceitos de geometria, álgebra, planilhas, gráficos, estatística e cálculo em uma única plataforma, atendendo a diferentes níveis de ensino. Apesar da utilidade desses softwares, a maioria dos tópicos matemáticos ainda é apresentada aos alunos de forma estática, através de livros ou outros materiais textuais. Embora essa abordagem seja eficaz para transmitir informações, a utilização de animações se mostra mais eficiente para demonstrar e explicar os processos de construção do conhecimento.

Estudos como o de Taylor, Pountney e Malabar (2007) comprovam que a apresentação de conceitos como adição e multiplicação de matrizes através de animações resulta em um aprendizado mais eficaz em comparação com a apresentação estática. Nesse estudo, a adição de duas matrizes 2×2 é animada, mostrando os números de cada matriz se movendo para suas posições na matriz resultante, com o sinal de soma sendo inserido e, por fim, o resultado é exibido.

Com os avanços tecnológicos, a criação de animações tornou-se mais acessível. No entanto, a produção de animações sofisticadas e de alta qualidade ainda representa um desafio para aqueles que não são especialistas em edição. Nesse contexto, podemos destacar as animações voltadas para o ensino da matemática, como na série "O Universo Mecânico", que utiliza animações para ilustrar fórmulas, matrizes e figuras geométricas, um diferencial notável para a época. Produzida em 1985, essa série já explorava objetos animados como recurso para aprimorar a aprendizagem matemática. Seguindo essa mesma proposta, a biblioteca Manim (do inglês, *Mathe-*

matical Animation Engine), uma ferramenta voltada para a animação de elementos matemáticos, tem possibilitado avanços significativos nos últimos anos.

Com a capacidade de explorar o pensamento computacional e proporcionar a visualização de conceitos na mais alta qualidade de animação, o Manim se mostra como uma grande ferramenta para as práticas pedagógicas de matemática. Essa biblioteca oferece recursos capazes de transformar conceitos abstratos em representações dinâmicas, tornando mais possível o engajamento dos alunos e facilitando a compreensão de conteúdos complexos.

Por ser uma ferramenta de código aberto, o Manim é gratuito e flexível, permitindo aos educadores personalizar os conteúdos de acordo com suas necessidades pedagógicas. No entanto, seu uso exige conhecimentos prévios de programação, especialmente em Python, o que pode representar um desafio para quem não tem familiaridade com o ambiente da informática. Além disso, a elaboração de animações mais avançadas pode exigir um investimento considerável de tempo e esforço, além de um computador com configurações apropriadas para executar tarefas mais complexas. Apesar das dificuldades elencadas, o Manim se destaca por oferecer uma ampla base de instruções e exemplos disponíveis em seu site oficial, facilitando o aprendizado tanto para iniciantes quanto para usuários mais experientes. Dessa forma, mesmo quem está começando encontra suporte para explorar todos os recursos da ferramenta.

Para Kishimoto e Coluci (2023), a biblioteca Manim serviu como ferramenta para a produção de animações de temas matemáticos e físicos do ensino médio e superior. Eles produziram ainda um manual em português para que iniciantes possam desenvolver suas próprias animações. Além deles, Castillo e Sánchez (2025) deram vida a um projeto chamado AM², que tem como objetivo propor animações matemáticas e mostrar as potencialidades pedagógicas do Manim.

Atualmente, os trabalhos realizados com o Manim concentram-se majoritariamente na criação de animações matemáticas. Entretanto, a exploração do pensamento computacional durante o processo de construção dessas animações é uma abordagem que potencializa o desenvolvimento de habilidades essenciais, como a resolução de problemas, o raciocínio lógico e a criatividade. Essa estratégia não apenas promove uma compreensão mais sólida dos conteúdos, mas também oferece uma oportunidade de integrar tecnologia e pensamento computacional na resolução de problemas práticos, proporcionando mais qualidade na formação dos alunos.

Este trabalho diferencia-se da literatura existente por estar estruturado a partir da matriz curricular do SAEPE, em consonância com a Base Nacional Comum Curricular (BNCC), o que assegura maior conectividade com a formação e alinhamento pedagógico da educação básica. Além disso, os materiais produzidos apresentam caráter aberto e dinâmico, uma vez que os códigos desenvolvidos estão disponíveis

em repositório online, permitindo que sejam editados, adaptados e ampliados por outros professores e pesquisadores, o que não ocorre com a maioria dos projetos feitos com essa biblioteca.

1.3 Objetivos

1.3.1 Objetivo geral

Explorar o uso da biblioteca Manim como ferramenta didática no ensino da matemática, promovendo o desenvolvimento do pensamento computacional e a compreensão de conceitos por meio da visualização e resolução de exercícios inspirados em avaliações externas, como o ENEM, SAEPE e SAEB.

1.3.2 Objetivos específicos

- Compreender como o pensamento computacional pode ser desenvolvido no ensino de matemática por meio da programação de animações;
- Elaborar um tutorial sucinto sobre a utilização da linguagem de programação Python e da biblioteca Manim;
- Criar animações matemáticas com a biblioteca Manim que representem conceitos e soluções de problemas presentes no SAEPE visando facilitar a aprendizagem por meio do raciocínio visual.

Esta dissertação tem como produto um conjunto de animações matemáticas desenvolvidas com a biblioteca Manim, recomendadas para uso pedagógico em sala de aula. As animações são voltadas, principalmente, para a resolução e análise de questões extraídas de avaliações externas com o objetivo de apoiar o trabalho docente e enriquecer a experiência de aprendizagem dos estudantes. Todo o material produzido está disponível no endereço:

`https://onmat.github.io/SAEPE-Animado/`.

1.4 Organização do trabalho

Além deste, esta dissertação consiste de outros quatro capítulos. O Capítulo 2 apresenta os fundamentos da linguagem Python, destacando os conceitos essenciais para a criação de scripts voltados à visualização matemática. São abordadas suas estruturas básicas como variáveis, funções, condicionais, laços de repetição e manipulação de

bibliotecas. O capítulo seguinte trata especificamente da biblioteca Manim, explorando sua estrutura, principais objetos gráficos, métodos de animação, estruturação de projetos e exemplos iniciais de uso. A posterior aplicação desses conceitos na construção de animações educativas acontece no Capítulo 4, com foco em conteúdos matemáticos da matriz do SAEPE. Nele, são apresentadas o passo a passo para a construção de vídeos com a resolução comentada de questões e explicações teóricas. Por fim, no Capítulo 5, são apresentadas as considerações finais.

Capítulo 2

Uma introdução ao Python

Neste capítulo, são apresentadas noções básicas sobre a linguagem de programação Python, cuja compreensão é indispensável para o uso adequado da biblioteca Manim, empregada nos capítulos seguintes. O Python é uma linguagem de alto nível, interpretada e com sintaxe simplificada, características que a tornam acessível tanto a iniciantes quanto a programadores experientes. Seu uso é amplamente difundido em diversas áreas, como automação, ciência de dados, inteligência artificial, desenvolvimento web, e, mais recentemente, na educação e no ensino de matemática.

Desenvolvida por Guido van Rossum e lançada oficialmente em 1991, o Python destacou-se desde o início por sua legibilidade e clareza sintática. A linguagem foi projetada com foco na produtividade e na facilidade de manutenção, permitindo que o programador dedique mais atenção à lógica do problema do que a aspectos técnicos da linguagem. Desde então, o Python tem evoluído constantemente, incorporando novas funcionalidades e consolidando-se como uma das linguagens mais populares do mundo.

A crescente adoção de Python pela comunidade científica se deu, sobretudo, pela sua capacidade de integração com outras linguagens e ferramentas, além de sua vasta gama de bibliotecas voltadas à análise de dados, modelagem matemática e computação simbólica. Essa característica permitiu que cientistas e pesquisadores concentrassem seus esforços nas tarefas analíticas, reduzindo a complexidade da implementação computacional. Além disso, o engajamento da comunidade de desenvolvedores, extremamente ativa e colaborativa, tem sido essencial para a constante atualização e expansão do ecossistema Python, com destaque especial para as áreas de ciência de dados, inteligência artificial e educação.

O objetivo deste capítulo é fornecer ao leitor uma base sólida para leitura, escrita e interpretação de códigos em Python, essencial para o desenvolvimento de animações matemáticas utilizando a biblioteca Manim. Para isso, iniciaremos com uma explicação sobre o fluxo de desenvolvimento em Python, abordando como o código é construído e testado progressivamente. Em seguida, exploraremos os con-

ceitos fundamentais de variáveis e tipos de dados, estruturas de repetição e controle de fluxo, finalizando com uma introdução à programação orientada a objetos, com foco na criação de classes e objetos, elemento central na organização dos projetos desenvolvidos com a Manim.

2.1 Fluxo de desenvolvimento

Antes de iniciar a escrita do código, o primeiro passo é garantir que o ambiente de programação esteja devidamente configurado. Para programar em Python, é necessário fazer a instalação da linguagem na máquina. Após a instalação, recomenda-se utilizar uma IDE (do inglês, *Integrated Development Environment*), ou ambiente de desenvolvimento integrado, que facilita a escrita, execução e depuração do código. Exemplos populares são o Visual Studio Code, o PyCharm e o próprio IDLE, que já vem junto com a instalação do Python. Essas ferramentas oferecem recursos como realce de sintaxe, autocompletamento, execução integrada e terminais interativos.

Com o ambiente pronto, o processo começa pela escrita do código-fonte, que é o conjunto de instruções escritas em uma linguagem de alto nível, como a Python. No caso do Python, esse código é interpretado, ou seja, existe um interpretador que lê o código-fonte linha a linha, traduzindo-o em tempo real para operações que o computador consiga executar. Por outro lado, linguagens compiladas como C ou C++ seguem um caminho diferente: o código-fonte é processado por um compilador, que traduz todo o programa para um código objeto em linguagem de baixo nível (linguagem máquina). Esse código objeto é então executado diretamente pelo computador, produzindo a saída desejada. Apesar dessas diferenças técnicas, o fluxo de desenvolvimento, a forma como o programador escreve, testa, corrige e melhora o código, é comum a ambas as abordagens e fundamental para a construção de software.

2.1.1 Primeiro programa: *Hello World.*

Uma prática recorrente ao explorar uma nova linguagem de programação é a impressão da frase "Hello World", que se tornou um marco simbólico para iniciantes. Embora essa tradição seja amplamente adotada por desenvolvedores, não é uma regra formal, mas sim uma abordagem introdutória comum. A Figura 2.1 mostra como isto é feito em Python.

Figura 2.1: Primeiro programa: Hello World.

Fonte: Elaborado pelo autor.

2.1.2 Execução e erros

A execução de um programa em Python segue uma ordem sequencial: o interpretador lê o código linha por linha, executando as instruções conforme aparecem. Se não houver erros, o programa termina normalmente. Contudo, durante esse processo, é comum encontrar dificuldades que geram interrupções. Essas dificuldades são os erros, e compreendê-los é essencial para um bom desenvolvimento.

Os erros em programação podem ser classificados em três tipos principais:

- **Erros de Sintaxe**: São aqueles relacionados à estrutura do código. O Python possui regras rígidas sobre como as instruções devem ser escritas, e qualquer violação dessas regras impede a execução do programa. Por exemplo, esquecer de fechar um parêntese ou usar incorretamente os dois pontos ao fim de uma instrução `if` gera erro de sintaxe. Ao tentar executar, o interpretador exibirá uma mensagem de erro informando a falta do parêntese, e o programa não será executado.

- **Erros de Execução (Runtime Errors)**: Estes erros só aparecem enquanto o programa está rodando, quando ocorre alguma situação inesperada, como tentar dividir um número por zero ou acessar uma posição inexistente em uma lista. Esses erros são chamados de exceções e fazem o programa parar abruptamente.

- **Erros de Semântica (Erros de Lógica)**: Estes erros são mais sutis. O programa não apresenta erros sintáticos nem de execução, mas não realiza a tarefa desejada porque a lógica está incorreta. O código roda normalmente, porém os resultados são errados.

A *depuração* (*debugging*) é o processo de localizar e corrigir erros no código. As IDEs modernas possuem ferramentas integradas de depuração que permitem pausar o programa, inspecionar variáveis e avançar passo a passo. Outro aspecto fundamental da depuração é a validação experimental: testar o programa com diferentes entradas para garantir que ele se comporta corretamente em todos os casos previstos. Esse método é especialmente importante para detectar erros de lógica.

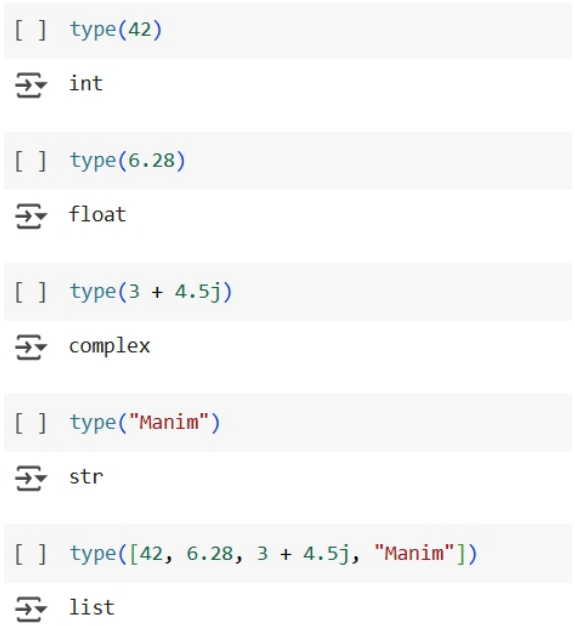
Para muitas pessoas, programar está intimamente ligado ao processo de depuração, visto que desenvolver um programa envolve construir gradualmente um código que funcione corretamente. A prática comum é iniciar com um programa simples que já execute alguma função básica e, a partir daí, realizar pequenas modificações, testando e corrigindo erros à medida que se avança. Dessa forma, mantém-se sempre uma versão funcional do programa durante o desenvolvimento.

2.2 Variáveis e tipos de dados

As variáveis constituem um dos conceitos fundamentais em qualquer linguagem de programação. Elas representam espaços de memória destinados ao armazenamento de valores que podem ser manipulados ao longo da execução de um programa. Para que o código seja interpretado corretamente, é essencial que a linguagem reconheça o tipo de dado associado a cada variável.

No caso do Python, essa identificação é feita de forma dinâmica, ou seja, o tipo da variável é determinado automaticamente com base no valor atribuído. Ainda assim, é possível verificar explicitamente o tipo de uma variável utilizando a função `type`. Essa função recebe o nome da variável entre parênteses e retorna a classe à qual o valor pertence. O comando pode ser utilizado tanto no terminal interativo quanto em um script executado por uma IDE, como mostra a Figura 2.2.

Figura 2.2: Identificação do tipo de variáveis.



Fonte: Elaborado pelo autor.

Como é possível observar por meio da função `type`, diferentes valores são automaticamente classificados pelo interpretador Python conforme seu tipo de dado. Por exemplo, o número `42` é identificado como do tipo `int`, representando um valor inteiro. Já o valor `6.28` pertence ao tipo `float`, que indica um número com ponto flutuante. Quando se utiliza a notação `3 + 4.5j`, o Python reconhece esse valor como `complex`, ou seja, um número complexo. A cadeia de caracteres `"Manim"` é classificada como `str` (*string*), e o último exemplo, contendo uma sequência de valores entre colchetes, é do tipo `list`, representando uma lista, estrutura que será abordada mais a diante.

Para declarar uma variável em Python, basta atribuir-lhe um valor. Devido ao seu alto nível de abstração, a linguagem não exige a declaração explícita de tipos, como ocorre em linguagens como C, onde é necessário definir previamente se uma variável é `int, float, char`, entre outros. Em Python, a inferência de tipo ocorre automaticamente durante a atribuição.

Apesar dessa simplicidade, há algumas regras para a nomeação de variáveis. Não é permitido iniciar identificadores com números, nem utilizar caracteres especiais ou espaços. Além disso, palavras reservadas da linguagem, as chamadas palavras-chave, não podem ser usadas como nomes de variáveis, pois já possuem significados específicos no funcionamento do interpretador. A lista completa dessas palavras pode ser consultada executando os comandos `import keyword` e `print(keyword.kwlist)`, em linhas separadas (Figura 2.3).

Figura 2.3: Lista de palavras-chave reservadas do Python.

False	None	True	and
as	assert	async	await
break	class	continue	def
del	elif	else	except
finally	for	from	global
if	import	in	is
lambda	nonlocal	not	or
pass	raise	return	try
while	with	yield	

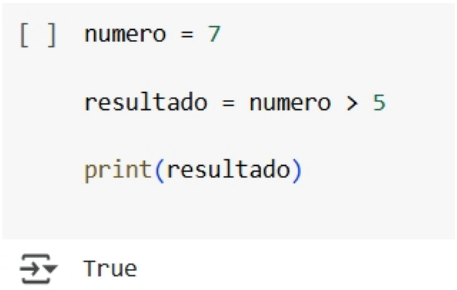
Fonte: Elaborado pelo autor.

Por fim, é importante mencionar que algumas variáveis podem ter o tipo `bool`, que representa valores boolianos. Essas variáveis só podem assumir dois estados: `True` (verdadeiro) ou `False` (falso).

No exemplo ilustrado na Figura 2.4, a variável `numero` recebe o valor 7. A seguir, é atribuída à variável `resultado` o resultado da expressão lógica `numero > 5`, que avalia se o valor armazenado em `numero` é maior que 5. Essa expressão retorna um

valor booliano, `True` caso a condição seja verdadeira, ou `False` caso contrário. Por fim, a função `print` exibe o valor de `resultado`, mostrando que, como 10 é maior que 5, o resultado da avaliação é `True`.

Figura 2.4: Exemplo de avaliação de uma expressão booliana.



Fonte: Elaborado pelo autor.

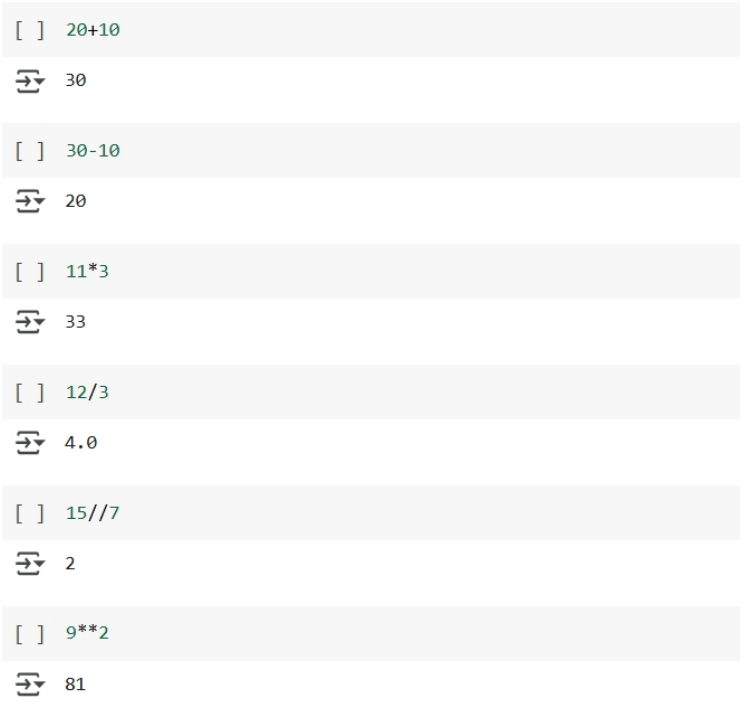
2.3 Expressões aritméticas

Outro tema fundamental na programação são as expressões utilizadas em cada linguagem. Em Python, as expressões aritméticas seguem padrões semelhantes aos de outras linguagens, facilitando seu entendimento.

Os operadores básicos são: +, -, *, / e %, que realizam, respectivamente, adição, subtração, multiplicação, divisão e cálculo do resto da divisão entre dois números.

Além desses, Python inclui duas operações adicionais: a exponenciação, representada por **, e a divisão inteira, representada por //, que retorna apenas a parte inteira do resultado da divisão. Vale destacar que a divisão comum (/) sempre produz um valor do tipo `float`. Na Figura 2.5, são mostrados exemplos dessas operações.

Figura 2.5: Expressões aritméticas em Python.



Fonte: Elaborado pelo autor.

2.4 Expressões lógicas

Seguindo o raciocínio das expressões aritméticas, as expressões lógicas também desempenham um papel central na construção de algoritmos. Em diversas situações, como em estruturas condicionais e laços de repetição, essas expressões são utilizadas com ainda mais frequência do que as aritméticas, tornando-se essenciais para a lógica da programação.

No contexto da linguagem Python, destacam-se três grupos de operadores relacionados à lógica: os operadores relacionais, que comparam valores; os operadores lógicos, que combinam expressões booleanas; e os operadores bit a bit (bitwise), que operam diretamente sobre representações binárias. Todos esses elementos estão organizados na Tabela 2.1.

Tabela 2.1: Operadores lógicos, relacionais e bitwise em Python.

Operadores	**Descrição**
`and`	Retorna um valor verdadeiro se, e somente se, receber duas expressões verdadeiras.
`or`	Retorna um valor falso se, e somente se, receber duas expressões falsas.
`not`	Retorna verdadeiro se receber uma expressão falsa, e vice-versa.
`is`	Retorna verdadeiro se, e somente se, receber duas expressões cujos valores são iguais.
`in`	Retorna verdadeiro se, e somente se, receber um valor contido numa lista, tupla, dicionário, etc.
`X << Y`	Retorna `X` com os bits deslocados à esquerda por `Y` lugares.
`X >> Y`	Retorna `X` com os bits deslocados à direita por `Y` lugares.
`~ X`	Retorna o complemento de `X`. É equivalente a: `-X - 1`.
`X ^ Y`	Bitwise exclusivo (XOR a cada bit). O bit da saída é 0 se o bit da entrada for igual ao de `Y`; caso contrário, é 1.
`X == Y`	Retorna verdadeiro se, e somente se, `X` for igual a `Y`.
`X != Y`	Retorna verdadeiro se, e somente se, `X` for diferente de `Y`.
`X > Y`	Retorna verdadeiro se, e somente se, `X` for maior que `Y`.
`X < Y`	Retorna verdadeiro se, e somente se, `X` for menor que `Y`.
`X >= Y`	Retorna verdadeiro se, e somente se, `X` for maior ou igual a `Y`.
`X <= Y`	Retorna verdadeiro se, e somente se, `X` for menor ou igual a `Y`.

Fonte: Elaborada pelo autor.

Na Figura 2.6, temos um exemplo de aplicação do operador `in`. A primeira parte do código define uma `lista` de frutas e uma variável `fruta`. A expressão `fruta in lista` verifica a presença de `'banana'` na lista, resultando em `True`, que é então exibido. Em seguida, temos a verificação se o número `10` está no `range(1,9)`. Como `range(1,9)` representa os números de 1 a 9, e 10 não está incluído, o resultado da operação é `False`, que também tem seu valor exibido.

Figura 2.6: Exemplo de utilização do operador `in`.

```
lista = ['uva',"manga",'banana']
fruta = 'banana'
resultado = fruta in lista
print(resultado)

print()

numero = 10
intervalo = numero in range(1,9)
print(intervalo)
```

```
True

False
```

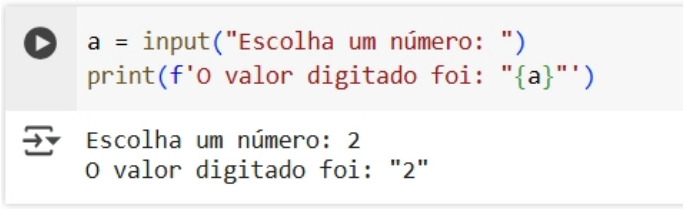
Fonte: Elaborado pelo autor.

2.5 Entrada e saída de dados

Em Python, a forma mais simploes de entrada de dados pelo usuário é realizada por meio da função `input`. Essa função permite capturar informações inseridas pelo teclado durante a execução do programa, podendo ser atribuídas a variáveis previamente definidas. Vale ressaltar que todos os dados obtidos por meio de `input` são, por padrão, armazenados como valores do tipo `str`. Portanto, quando se deseja trabalhar com outros tipos de dados, como inteiros (`int`), números reais (`float`) ou números complexos (`complex`), é necessário realizar a conversão explícita. Para isso, basta utilizar o construtor correspondente ao tipo desejado. Por exemplo, para converter um dado inserido pelo usuário em um tipo numérico específico, utiliza-se a sintaxe `tipodesejado(input())`. Assim, para transformar a entrada em um número inteiro, escreve-se `int(input())`; se a conversão desejada for para um número em ponto flutuante, basta utilizar `float(input())`.

Além disso, a função `input` permite exibir uma mensagem ao usuário no momento da entrada de dados. Para isso, basta passar a mensagem desejada como argumento da função, entre aspas. Por exemplo: `input("Digite um número: ")`. A Figura 2.7 contém um exemplo que ilustra os casos mencionados anteriormente.

Figura 2.7: Exemplo de utilização da função `input` para a entrada de dados.



Fonte: Elaborado pelo autor.

Para realizar a conversão de variáveis entre os diferentes tipos disponíveis em Python, aplica-se o mesmo princípio apresentado anteriormente. Essa lógica é válida tanto para a conversão de uma variável do tipo `str` para os tipos `int`, `float` ou `complex`, quanto para o processo inverso, converter de `int`, `float` ou `complex` para `str`.

No que se refere à saída de dados, a forma mais elementar e amplamente utilizada em Python é a função `print`. Essa função possibilita ao programador exibir informações diretamente na tela, oferecendo uma maneira prática e direta de se comunicar com o usuário.

Seu funcionamento é bastante intuitivo: para imprimir um texto, basta colocá-lo entre aspas (simples ou duplas). Para exibir o valor de uma variável, é suficiente referenciar seu nome. Quando se deseja combinar texto e variáveis na mesma chamada da `print`, a forma de concatenação dependerá do tipo da variável. Caso a variável seja do tipo `str`, utiliza-se o operador `+` para realizar a concatenação. Já para variáveis de outros tipos, como `int` ou `float`, é necessário separá-las com vírgulas para que sejam interpretadas corretamente e convertidas automaticamente em texto na saída. Na Figura 2.8, apresenta-se um exemplo prático de uso da função `print`.

Figura 2.8: Exemplo de utilização da função `print` para a saída de dados.

Fonte: Elaborado pelo autor.

Como os valores atribuídos às variáveis podem ser convertidos para o tipo `str`, podemos, em seguida, concatená-los por meio do operador `+`. A concatenação de

strings consiste, de maneira prática, na união de duas ou mais cadeias de caracteres em uma única sequência. Por exemplo, considere as atribuições `a = 'Amo'` e `b = ' Python'`. Ao definir `c = a + b`, a variável `c` armazenará o valor `'Amo Python'`.

Também é possível repetir uma cadeia de caracteres. Isto é feito utilizando-se o operador de multiplicação. Nesse caso, escreve-se `n * cadeia`, em que `n` representa o número inteiro de repetições desejadas e `cadeia` é a sequência de caracteres a ser repetida.

No código da Figura 2.9, duas variáveis do tipo `str` são manipuladas. A atribuição `c = a + b` realiza a concatenação, unindo seus valores em uma única variável. Já na variável `d`, aplicamos a repetição, duplicando o conteúdo de `a` e repetindo `b` cinco vezes, resultando em uma cadeia com sete palavras.

Figura 2.9: Concatenação e repetição de cadeias de caracteres.

```
a="Amo"
b=" Python"
c= a + b
print(c)

print()

d= 2*a + 5*b
print(d)
```

```
Amo Python

AmoAmo Python Python Python Python Python
```

Fonte: Elaborado pelo autor.

Uma forma mais eficiente de utilizar a função `print` é por meio da interpolação de variáveis, que permite incorporar valores diretamente dentro da estrutura do texto, sem a necessidade de múltiplas concatenações com o operador `+`. Esse método não apenas torna o código mais legível, mas também melhora seu desempenho em situações que exigem manipulação dinâmica de strings. Para utilizá-lo, é essencial conhecer previamente o tipo da variável e aplicar o operador `%` seguido da letra correspondente ao formato adequado. A Tabela 2.2 apresenta os símbolos utilizados nesse processo de interpolação.

Tabela 2.2: Identificadores de interpolação usados pela função `print`.

Símbolo	Formato de Saída
`%s`	Cadeia de caracteres
`%d`	Inteiro
`%f`	Real
`%o`	Octal
`%x`	Hexadecimal
`%e`	Real em notação científica
`%%`	Sinal de porcentagem

Fonte: Elaborada pelo autor.

Na Figura 2.10 é exemplificado o uso desses símbolos na interpolação de strings com a função `print`. A variável `s1` define um padrão de impressão com marcadores de posição (`%s` para `str`, `%d` para inteiro, `%.1f` para ponto flutuante com uma casa decimal e `%%` para o símbolo de porcentagem). As variáveis `nome`, `paginas` e `porcentagem` armazenam os dados a serem interpolados. O operador `%` é então utilizado para inserir os valores dessas variáveis nos marcadores de posição em `s1`, resultando na cadeia de caracteres final `s2`, que é posteriormente exibida pela função `print`.

Figura 2.10: Interpolação de cadeias de caracteres com a função `print`.

```
s1 = ("%s, o livro tem %d páginas e %.1f%% de material reciclado.")
nome = 'Ícaro'
paginas = 234
porcentagem = 12.5
s2 = s1 %(nome,paginas,porcentagem)

print(s2)
```

```
Ícaro, o livro tem 234 páginas e 12.5% de material reciclado.
```

Fonte: Elaborado pelo autor.

2.6 Estruturas de repetição

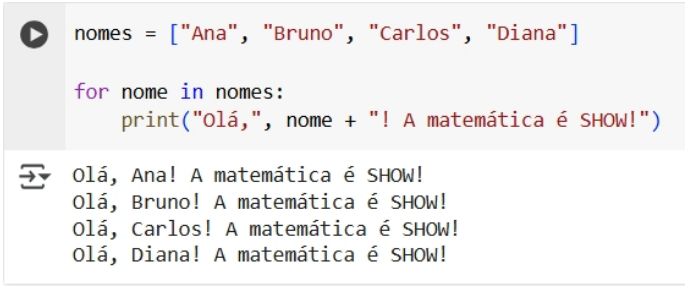
As linguagens de programação oferecem mecanismos que permitem a execução repetida de determinadas instruções, de forma controlada e eficiente. Tais mecanismos são conhecidos como *estruturas de repetição* ou *laços* (*loops*, em inglês). Essas estru-

turas são fundamentais na construção de algoritmos, pois possibilitam automatizar tarefas que exigem múltiplas execuções com base em critérios previamente definidos.

Em Python, as duas principais estruturas de repetição são os comandos o `for` e o `while`. Ambas desempenham papéis essenciais no controle do fluxo de execução de programas, especialmente em situações em que se deseja processar coleções de dados, realizar contagens, ou manter a execução de um bloco de código enquanto determinada condição lógica for satisfeita.

O comando `for` é amplamente utilizado quando o número de repetições é conhecido ou determinado por uma sequência de elementos. Trata-se de um laço iterativo, no qual uma variável percorre, elemento por elemento, uma coleção (como listas, sequências, cadeias de caracteres ou intervalos numéricos definidos pela função `range`). Esse tipo de repetição é especialmente útil quando se deseja aplicar uma mesma operação sobre todos os elementos de uma coleção ou controlar a execução de comandos com base em uma contagem previamente estabelecida. Na Figura 2.11, temos um exemplo de código utilizando um laço com o `for`.

Figura 2.11: Exemplo de laço com o comando `for`.



Fonte: Elaborado pelo autor.

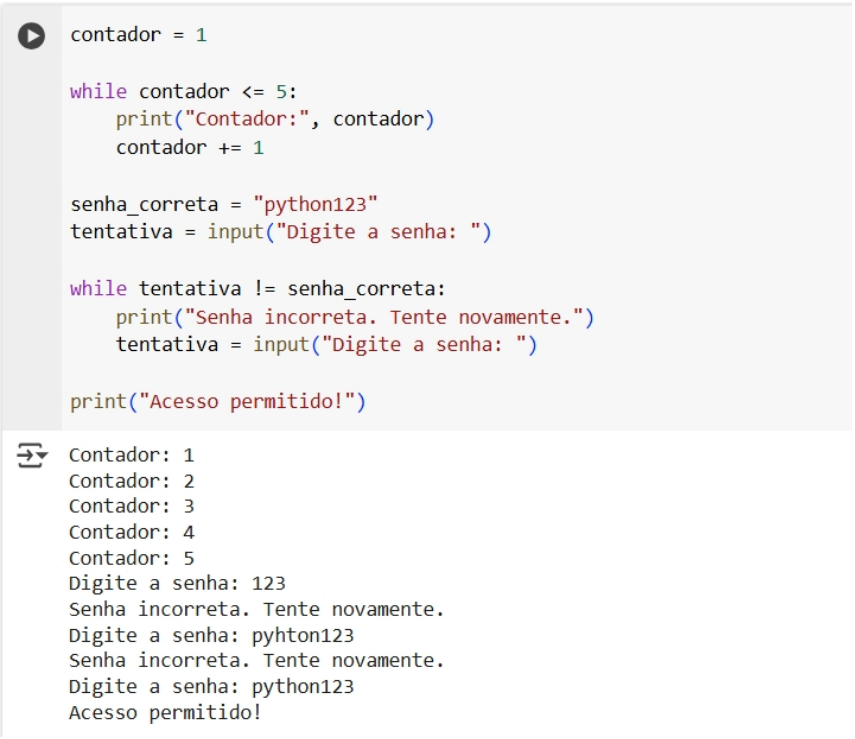
Neste exemplo, criamos uma lista chamada `nomes` com quatro elementos. O comando `for nome in nomes` faz com que a variável `nome` assuma, a cada iteração, o valor de um dos elementos da lista. A função `print` é chamada a cada volta do laço, exibindo uma mensagem personalizada para cada nome.

Já a estrutura `while` é caracterizada por ser uma repetição condicionada. Isso significa que a execução do bloco de código depende da veracidade de uma expressão lógica, a qual é avaliada antes de cada iteração. Enquanto a condição for verdadeira, o código dentro do laço continuará a ser executado. Esta estrutura é adequada para situações em que o número de repetições não é conhecido de antemão, e se deseja manter a repetição até que uma determinada condição deixe de ser satisfeita.

Embora os dois sejam parecidos, há bastante diferença no modo como os laços `for` e `while` controlam a repetição. Enquanto o `for` percorre sequências com um número definido de passos, o `while` depende da veracidade contínua de uma condição lógica, podendo executar um número indefinido de vezes.

Veja na Figura 2.12 o uso do `while`. Na primeira parte, da 1ª até a 4ª linha, temos uma estrutura `while` simples, na qual a variável `contador` é inicializada com o valor `1` e, enquanto for menor ou igual a `5`, o bloco de instrução será executado. A cada iteração, imprime-se o valor atual do contador, e em seguida a variável é incrementada. Enquanto que a segunda parte do código, o comando `while` é utilizado para controlar o fluxo de entrada de dados até que o usuário insira a senha correta. A variável `senha_correta` armazena a senha válida, enquanto a variável `tentativa` guarda a entrada do usuário. Enquanto a condição `tentativa != senha_correta` for verdadeira, o programa continuará solicitando a senha. Assim que o usuário acertar, o laço é interrompido e a mensagem de acesso permitido é exibida. Este é um exemplo clássico de laço de repetição baseado em validação de entrada.

Figura 2.12: Exemplo de laço com o comando `while`.



```
contador = 1

while contador <= 5:
    print("Contador:", contador)
    contador += 1

senha_correta = "python123"
tentativa = input("Digite a senha: ")

while tentativa != senha_correta:
    print("Senha incorreta. Tente novamente.")
    tentativa = input("Digite a senha: ")

print("Acesso permitido!")
```

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5
Digite a senha: 123
Senha incorreta. Tente novamente.
Digite a senha: pyhton123
Senha incorreta. Tente novamente.
Digite a senha: python123
Acesso permitido!
```

Fonte: Elaborado pelo autor.

2.7 Controle de fluxo

Uma vez compreendidos os operadores lógicos e as estruturas de repetição, é fundamental agora explorar as estruturas condicionais ou de controle de fluxo. Com esses elementos, o programa adquire a capacidade de tomar decisões e seguir diferentes caminhos de execução, com base em verificações lógicas.

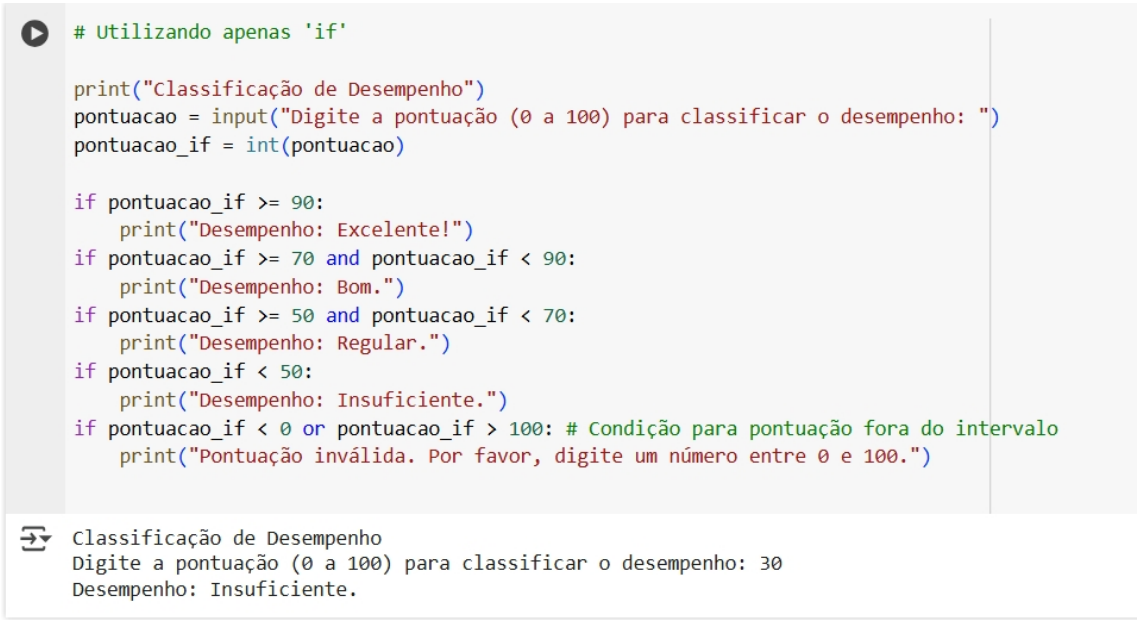
Em Python, as estruturas condicionais mais simples são representadas pelo `if` e o `else`. O `if` introduz um bloco de código que será executado somente se uma

condição específica for verdadeira, enquanto o `else` define um bloco alternativo a ser executado caso a condição do `if` seja falsa. Além disso, a linguagem oferece a ramificação `elif`, uma abreviação de "else if", que possibilita a inserção de múltiplas condições intermediárias de forma sequencial e lógica, otimizando a legibilidade e a estrutura do código.

Para ilustrar a aplicação prática de estruturas de decisão, considere os códigos-fonte das Figuras 2.13, 2.14 e 2.15. Todos eles realizam a mesma tarefa utilizando abordagens distintas: classificam o nível de desempenho de um usuário com base em uma pontuação informada pelo teclado.

Na Figura 2.13, o código emprega uma série de chamadas isoladas ao `if` para verificar as diferentes faixas de pontuação. Cada `if` opera de forma autônoma; ou seja, o programa verifica todas as condições listadas, independentemente de uma condição anterior já ter sido satisfeita. Isso pode levar a múltiplas mensagens sendo impressas se a lógica das condições não for mutuamente exclusiva.

Figura 2.13: Exemplo de utilização do `if`.



```
# Utilizando apenas 'if'

print("Classificação de Desempenho")
pontuacao = input("Digite a pontuação (0 a 100) para classificar o desempenho: ")
pontuacao_if = int(pontuacao)

if pontuacao_if >= 90:
    print("Desempenho: Excelente!")
if pontuacao_if >= 70 and pontuacao_if < 90:
    print("Desempenho: Bom.")
if pontuacao_if >= 50 and pontuacao_if < 70:
    print("Desempenho: Regular.")
if pontuacao_if < 50:
    print("Desempenho: Insuficiente.")
if pontuacao_if < 0 or pontuacao_if > 100: # Condição para pontuação fora do intervalo
    print("Pontuação inválida. Por favor, digite um número entre 0 e 100.")
```

```
Classificação de Desempenho
Digite a pontuação (0 a 100) para classificar o desempenho: 30
Desempenho: Insuficiente.
```

Fonte: Elaborado pelo autor.

Já na segunda abordagem, o código da Figura 2.14 emprega a combinação entre `if` e `else` de maneira aninhada. Primeiramente, uma condição `if` principal verifica a validade da pontuação. Se a pontuação estiver dentro da faixa aceitável (0 a 100), o bloco `else` correspondente é executado. Dentro deste `else`, uma nova sequência de `if` e `else` é aninhada para determinar a categoria de desempenho. Embora funcional, essa estrutura pode resultar em múltiplas indentações, tornando o código mais complexo visualmente à medida que o número de condições aumenta.

Figura 2.14: Exemplo de utilização do par `if-else` com aninhamento.

```python
#Classificação de Desempenho

print("Classificação de Desempenho")
pontuacao1 = input("Digite a pontuação (0 a 100) para classificar o desempenho: ")
pontuacao_ifelse = int(pontuacao1)

if pontuacao_ifelse < 0 or pontuacao_ifelse > 100:
    print("Pontuação inválida. Por favor, digite um número entre 0 e 100.")
else:
    if pontuacao_ifelse >= 90:
        print("Desempenho: Excelente!")
    else:
        if pontuacao_ifelse >= 70:
            print("Desempenho: Bom.")
        else:
            if pontuacao_ifelse >= 50:
                print("Desempenho: Regular.")
            else:
                print("Desempenho: Insuficiente.")
```

```
Classificação de Desempenho
Digite a pontuação (0 a 100) para classificar o desempenho: 60
Desempenho: Regular.
```

Fonte: Elaborado pelo autor.

Por fim, na versão ilustrada na Figura 2.15, o código contém a estrutura `if-elif-else`, que é a forma mais recomendada para lidar com condições mutuamente exclusivas em Python. O processo de avaliação é sequencial: o programa verifica a condição do `if` inicial; se esta for falsa, passa para o primeiro `elif`, e assim por diante. A grande vantagem é que, assim que uma condição (seja `if` ou `elif`) é avaliada como verdadeira, o bloco de código correspondente é executado, e as demais condições `elif` e o bloco `else` são ignorados. Isso reduz a quantidade de instruções executadas e torna o código mais claro, além de garantir que apenas uma saída será gerada para cada entrada válida. O bloco `else` final captura qualquer pontuação que não se enquadre nas categorias anteriores (neste caso, "Insuficiente") ou que seja inválida (se a validação estiver no `if` inicial).

Figura 2.15: Exemplo de utilização do trio `if-elif-else`.

```python
#Classificação de Desempenho

print("Classificação de Desempenho")
pontuacao2 = input("Digite a pontuação (0 a 100) para classificar o desempenho: ")
pontuacao_elif = int(pontuacao2)

if pontuacao_elif < 0 or pontuacao_elif > 100:
    print("Pontuação inválida. Por favor, digite um número entre 0 e 100.")
elif pontuacao_elif >= 90:
    print("Desempenho: Excelente!")
elif pontuacao_elif >= 70:
    print("Desempenho: Bom.")
elif pontuacao_elif >= 50:
    print("Desempenho: Regular.")
else:
    print("Desempenho: Insuficiente.")
```

```
Classificação de Desempenho
Digite a pontuação (0 a 100) para classificar o desempenho: 70
Desempenho: Bom.
```

Fonte: Elaborado pelo autor.

2.8 Classes

Em programação, seja em Python ou em outras linguagens, distinguimos os tipos de dados dos dados propriamente ditos. Podemos estabelecer um paralelo entre o conceito de forma e os tipos de dados, no qual cada dado concreto representa, ou, mais tecnicamente, instancia, um tipo específico.

Cada tipo de dado possui propriedades e comportamentos previamente definidos. Os números inteiros `int`, por exemplo, não possuem parte decimal, podem assumir valores positivos ou negativos e são manipulados por meio de operadores aritméticos. As cadeias de caracteres `str` são imutáveis e respondem a operações específicas, como concatenação ou fatiamento. As listas `list`, por sua vez, são mutáveis e permitem operações como inserção, remoção e expansão por concatenação.

Nesse contexto, as *classes* representam uma generalização do conceito de tipo de dado. Elas oferecem ao programador a possibilidade de definir seus próprios tipos, de acordo com as necessidades específicas de seu programa. Uma classe é responsável por definir as propriedades (ou atributos) e os comportamentos (ou métodos) que serão comuns a um conjunto de dados que ela representa.

Em Python, a definição de uma classe é realizada com a palavra-chave `class`, como se vê no exemplo da Figura 2.16, aonde se define um tipo para representar contas bancárias. Cada dado que corresponde a um tipo definido por uma classe é denominado *objeto*, ou ainda, uma *instância* dessa classe.

Figura 2.16: Exemplo de definição de uma classe em Python.

```python
class Conta:
    def __init__(self, numero, saldo=0):
        self.saldo = saldo
        self.numero = numero
```

Fonte: Elaborado pelo autor.

A palavra-chave `def` é utilizada para definir métodos, isto é, funções que pertencem a uma classe. Um desses métodos é o `__init__`, um método especial que é executado automaticamente quando um novo objeto é criado, denominado *construtor*.

O parâmetro `self` é obrigatório em todos os métodos de uma classe. Ele representa o próprio objeto que está sendo criado ou manipulado. Em outras linguagens, esse papel é assumido por um identificador implícito, como o `this` do C++, mas em Python é necessário informá-lo explicitamente. Por exemplo, na definição da classe para representar contas bancárias, as instruções que utilizam `self.numero` e `self.saldo` significam que esses atributos são armazenados no objeto sendo criado. Dessa forma, cada instância da classe terá seu próprio número e saldo, que são acessados e modificados com base no contexto daquele objeto específico.

É possível enriquecer uma classe definindo diversos atributos e métodos. Para o caso de uma conta bancária, outros métodos que poderiam ser definidos são, por exemplo, `resumo`, `saque` e `deposito`, conforme ilustrado na Figura 2.17. Observe que todos eles utilizam o `self` para acessar os dados da própria instância da classe.

Figura 2.17: Exemplo de definição de métodos em uma classe em Python.

```python
class Conta:
    def __init__(self, numero, saldo=0):
        self.saldo = saldo
        self.numero = numero

    def resumo(self):
        print("CC numero: %s  Saldo: %10.2f" % (self.numero, self.saldo))

    def saque(self, valor):
        if self.saldo >= valor:
            self.saldo -= valor

    def deposito(self, valor):
        self.saldo += valor
```

Fonte: Elaborado pelo autor.

A criação e manipulação de objetos de uma classe envolve a chamada do método construtor com os valores desejados e o uso dos métodos definidos para operar sobre seus atributos. Isto está ilustrado na Figura 2.18, onde a variável `c1` representa uma conta de número 1234 e saldo inicial igual a 100.

Figura 2.18: Instanciação de uma classe do tipo `Conta`.

```
class Conta:
    def __init__(self, numero, saldo=0):
        self.saldo = saldo
        self.numero = numero

    def resumo(self):
        print("CC numero: %s  Saldo: %10.2f" % (self.numero, self.saldo))

    def saque(self, valor):
        if self.saldo >= valor:
            self.saldo -= valor

    def deposito(self, valor):
        self.saldo += valor
c1 = Conta(1234, 100)
c1.resumo()
c1.saque(50)
c1.resumo()
c1.deposito(200)
c1.resumo()
```

```
CC numero: 1234  Saldo:     100.00
CC numero: 1234  Saldo:      50.00
CC numero: 1234  Saldo:     250.00
```

Fonte: Elaborado pelo autor.

Como vimos, ao desenvolver um sistema de controle de contas para uma rede bancária, é necessário armazenar informações específicas sobre cada conta. Para isso, define-se uma classe que representa o tipo de dado "conta bancária", agregando as informações essenciais (como número da conta e saldo) e os comportamentos básicos esperados (como depósito e saque). A vantagem dessa abordagem está na organização e no encapsulamento: os dados e as operações que dizem respeito à conta ficam agrupados de forma coesa, sendo definidos e manipulados em conjunto. Isso favorece a legibilidade, a manutenção e a reutilização do código.

A definição de classes é o aspecto central da programação orientada a objetos em Python. Essa estrutura permite ao programador criar tipos personalizados de dados, adaptados aos problemas que deseja resolver. É essa flexibilidade que torna as classes ferramentas tão poderosas na construção de sistemas computacionais.

Capítulo 3

A biblioteca Manim

A biblioteca *Manim* (do inglês, *Mathematical Animation Engine*) é uma ferramenta de código aberto desenvolvida originalmente em 2015 por Grant Sanderson, matemático e divulgador conhecido pelo canal 3Blue1Brown. Seu objetivo principal é permitir a criação de animações matemáticas com elevado rigor visual e clareza conceitual, utilizando a linguagem de programação Python como base para a construção e manipulação dos elementos gráficos animados.

Inicialmente elaborada para atender às necessidades do próprio autor na produção de vídeos educacionais, a biblioteca foi posteriormente disponibilizada publicamente no GitHub, tornando-se um projeto colaborativo. Atualmente, a versão mais utilizada é a *Manim Community Edition* (ManimCE), mantida por uma comunidade ativa que continuamente aprimora suas funcionalidades, compatibilidade e documentação. Na verdade, existem três versões principais da biblioteca: a ManimCE, que utiliza a biblioteca `pycairo` para renderização; a ManimCairo, versão original que também utiliza a `pycairo`; e a ManimGL, que emprega OpenGL para renderizar vídeos em tempo real.

O Manim oferece uma ampla gama de recursos voltados à visualização matemática programada: é possível desenhar funções, equações e expressões em LaTeX, criar objetos geométricos bidimensionais e tridimensionais, além de aplicar transformações, animações e agrupamentos visuais com precisão e controle total via código. Essa abordagem diferencia o Manim de outras ferramentas de animação, permitindo que o usuário combine estrutura lógica e intencionalidade pedagógica na construção de conteúdos visuais dinâmicos.

3.1 Instalação

O uso do Manim requer, previamente, a instalação e configuração de um ambiente de programação em Python (versão 3.8 ou superior). Disponível em `https://www.manim.community/`, o site oficial da comunidade reúne toda a documentação

da biblioteca, incluindo instruções detalhadas de instalação, exemplos de uso e o repositório com o código-fonte completo.

A instalação pode ser realizada em diferentes sistemas operacionais (Windows, Linux e macOS), e envolve, preferencialmente, o uso da ferramenta `uv`, que permite gerenciar o ambiente do projeto e instalar o Manim diretamente a partir do repositório oficial. Também são indicadas ferramentas opcionais, como o `Chocolatey` ou o gerenciador de pacotes `pip`, que podem facilitar o processo em sistemas Windows. Neste trabalho, adota-se a abordagem recomendada pela comunidade, utilizando a ferramenta `uv`.

3.1.1 A ferramenta `uv`

A ferramenta `uv` é um gerenciador de ambientes e dependências para Python. Seu propósito é substituir e unificar o uso das ferramentas tradicionais `pip` e `venv`, tornando o processo de configuração mais prático.

Para instalar o `uv`, precisamos usar o terminal. Se você usa Windows, o terminal mais recomendado é a PowerShell, que tem mais recursos que o tradicional prompt de Comando (CMD). No Linux e macOS, usamos o terminal padrão do sistema.

No Windows

Para abrir o PowerShell com as permissões necessárias, siga estes passos:

1. Pesquise por PowerShell na barra de busca do Windows;
2. Clique com o botão de opções para selecionar "executar como administrador";
3. Uma janela irá surgir perguntando se aceita que o aplicativo faça alterações, clique em "Sim".
4. Com a PowerShell pronto, copie e cole o comando abaixo e pressione *Enter* e aceite todas as permissões que ele venha a pedir na sequência.

```
powershell -ExecutionPolicy ByPass -c `
    "irm https://astral.sh/uv/install.ps1 | iex"
```

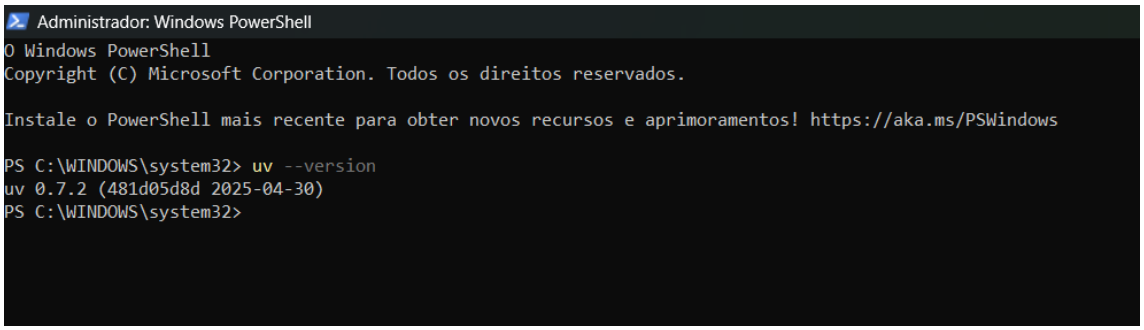
5. Feche e abra novamente a PowerShell.

Para confirmar que a instalação foi realizada com sucesso, execute:

```
uv --version
```

Se o terminal responder com um número de versão significa que está tudo certo, conforme a Figura 3.1.

Figura 3.1: Verificação da instalação do uv.



Fonte: Elaborado pelo autor.

No Linux e macOS

Abra o terminal e execute o comando:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

3.1.2 Criando um projeto

Agora que o uv está instalado, vamos criar uma pasta para o projeto Manim, onde ele funcionará de forma isolada. Primeiro, no terminal do PowerShell, mude para uma pasta de sua escolha, por exemplo:

```
cd C:\Users\usuario
```

Agora, execute no terminal o comando:

```
uv init manimations
```

Isto cria a pasta `manimations` e configura um ambiente Python exclusivo para esse projeto. Você pode escolher outro nome para a pasta, mas evite espaços e caracteres especiais.

Em seguida, entre na pasta criada com:

```
cd manimations
```

Agora estamos dentro do ambiente do projeto e podemos instalar o Manim.

3.1.3 Instalando o Manim

Para instalar o Manim, execute:

```
uv add manim
```

O comando baixa e instala o ManimCE na pasta do projeto, garantindo que ele funcione corretamente.

3.1.4 Instalando o LaTeX

O Manim permite criar animações com fórmulas matemáticas escritas em LaTeX, um sistema que gera textos matemáticos com alta qualidade visual. Para isso, é necessário ter instalado no computador uma distribuição do LaTeX. Se você não pretende usar fórmulas complexas, essa etapa pode ser ignorada, mas para este trabalho isso é obrigatório. Neste caso, procedemos do seguinte modo:

- No Windows, a recomendação é instalar o MiKTeX, disponível em `https://miktex.org/download`.
- No Linux (Debian ou Ubuntu), o LaTeX pode ser instalado pelo comando:

  ```
  sudo apt install texlive-full
  ```

- No macOS, a opção é instalar o MacTeX, disponível em `https://tug.org/mactex/`.

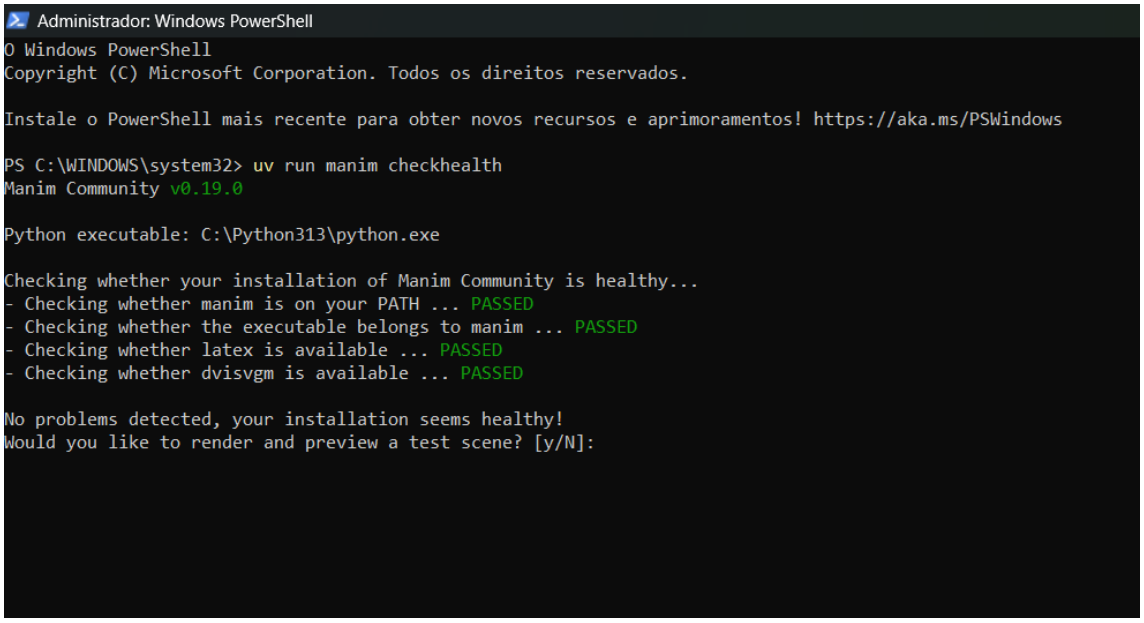
3.1.5 Verificação

Para testar se o Manim está instalado corretamente, execute no terminal o comando:

```
uv run manim checkhealth
```

Esse comando verifica o ambiente e informa se há algum problema a ser resolvido. Se estiver ocorrido tudo dentro da normalidade, ficará da mesma maneira que na Figura 3.2.

Figura 3.2: Verificação da instalação do Manim com o `checkhealth`.



Fonte: Elaborado pelo autor.

3.2 Hello World com o Manim

Continuando na pasta criada anteriormente, crie um subdiretório de nome `Teste manim` e nele crie um arquivo de nome `Teste_manim.py` contendo em seu interior o código:

```
from manim import *

class HelloWorld(Scene):
    def construct(self):
        texto = Text("Olá, mundo!")
        self.play(Write(texto))
        self.wait(1.5)
        self.play(FadeOut(texto))
```

No terminal, digite o seguinte comando para renderizar a animação:

```
uv run manim Teste_manim.py HelloWorld -pqm
```

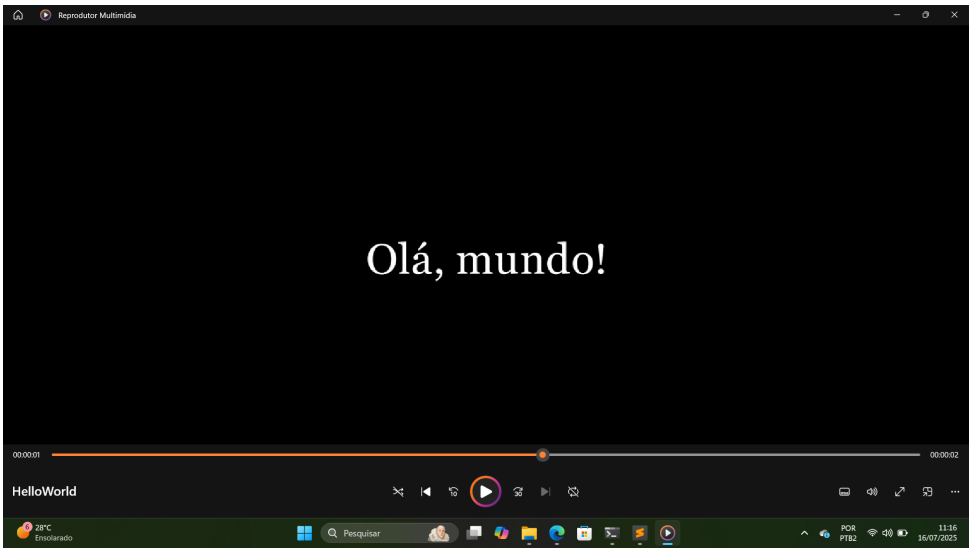
onde:

- `-pqm` indica que a renderização será feita com qualidade média (`qm`) e que o vídeo será automaticamente aberto (`p`);
- `HelloWorld` é o nome da classe principal da cena que será renderizada.

O Manim permite especificar outros níveis de qualidade de renderização por meio das opções: `-ql` (*low quality*), voltada para pré-visualizações rápidas com menor tempo de renderização; `-qh` (*high quality*), para vídeos de alta definição; e `-qk` (*4K resolution*), para renderizações em ultra-alta definição. A escolha da qualidade impacta diretamente no tempo de processamento e no tamanho do arquivo gerado.

Quando finalizado, automaticamente o player de vídeo do Windows irá reproduzir o arquivo, que deverá se parecer com a Figura 3.3.

Figura 3.3: Exemplo de um código Hello World usando o Manim.



Fonte: Elaborado pelo autor.

3.2.1 O código explicado

O comando:

```
from manim import *
```

é responsável por importar todos os módulos da biblioteca Manim. Isso significa que você poderá utilizar diretamente as classes e funções oferecidas por ela (`Scene`, `Text`, `Circle`, `Square`, `Write`, entre outras) sem precisar fazer referência ao módulo de origem a cada vez.

Na linha:

```
class HelloWorld(Scene):
```

estamos definindo uma cena a partir da classe chamada `HelloWorld`, que herdará da classe `Scene` do Manim métodos e atributos. Em termos simples, uma cena é o espaço onde tudo acontece. Nela inserimos os objetos, definimos os movimentos e organizamos a sequência de animações.

Para construir a cena, usamos a função `construct`, um método herdado da classe `Scene`. Ela é o ponto de entrada da cena. Tudo o que estiver dentro dessa função será executado quando a cena for renderizada.

Neste exemplo, a cena é formada somente pelo texto “Olá, mundo!”. Ao executar a linha:

```
texto = Text("Olá, mundo!")
```

estamos criando um objeto do tipo `Text` com o texto desejado. A classe `Text` é feita para renderizar texto comum (não matemático), utilizando por padrão a fonte Arial. Para fórmulas matemáticas, usamos outra classe que veremos adiante, a `MathTex`.

O método `play` é responsável por executar uma animação. No trecho:

```
self.play(Write(texto))
```

estamos utilizando a função `Write`, que é uma animação específica da Manim que simula o ato de escrever um texto na tela. Ela pode ser aplicada a objetos como `Text`, `MathTex`, `Circle`, `Square`, entre outros. Esse comando faz com que o texto apareça de forma animada, traçando suas letras uma a uma.

O comando `self.wait(1.5)` é utilizado para inserir uma pausa na execução da cena, mantendo todos os elementos visíveis na tela durante o intervalo especificado. O valor passado como argumento representa a duração da espera, em segundos.

A cena se encerra com a linha:

```
self.play(FadeOut(texto))
```

A animação da `FadeOut` tem como finalidade fazer com que um objeto gráfico desapareça gradualmente da cena. O efeito consiste na redução progressiva da opacidade do objeto até que ele se torne completamente transparente.

3.2.2 Estrutura de arquivos

Ao compilar um arquivo principal de uma animação em Manim, é automaticamente criada uma estrutura de arquivos que contém todas as informações envolvidas na renderização, conforme ilustrado na Figura 3.4. A seguir, descrevemos o papel de cada componente dessa estrutura.

Figura 3.4: Estrutura de arquivos gerada ao processar uma animação em Manim armazenada no arquivo `Teste_manim.py`.

Fonte: Elaborado pelo autor.

- **`__pycache__`**: pasta gerada automaticamente pelo interpretador Python para armazenar versões compiladas dos arquivos Python em bytecode, o que acelera a execução em chamadas futuras. No exemplo, contém o arquivo `Teste_simples.cpython-313.pyc`.
- **media**: diretório onde o Manim armazena todos os recursos gerados durante a renderização da cena. Esse diretório é subdividido em:
  - **texts**: armazena arquivos auxiliares em formato `.html` com informações relacionadas aos textos exibidos nas cenas.
  - **images/Teste_simples**: contém imagens (em geral, quadros ou componentes gráficos) geradas durante a construção da cena denominada `Teste_simples`.
  - **videos/Teste_simples/720p30**: diretório onde o vídeo final renderizado é armazenado com resolução de `720p` a `30` quadros por segundo. Dentro dele, temos:
    - **HelloWorld.mp4**: arquivo de saída principal da cena renderizada.

    - **partial_movie_files**: pasta contendo os trechos parciais do vídeo que são temporariamente gerados antes da concatenação final. No interior dela:
        - **HelloWorld**/: armazena os arquivos `video1.mp4`, `video2.mp4`, `video3.mp4`, entre outros, cada qual correspondente a um segmento da animação.
        - **partial_movie_files_list.txt**: arquivo de texto que especifica a ordem em que os vídeos parciais devem ser concatenados para formar o vídeo final.

Essa organização permite ao usuário do Manim depurar e acessar componentes específicos de uma cena, facilitando tanto a renderização quanto o reaproveitamento de imagens ou trechos de vídeo em outras possíveis produções.

3.3 Classes e métodos básicos

Após apresentar o funcionamento básico do Manim, é possível aprofundar a compreensão sobre os principais elementos que compõem essa biblioteca. Nesta seção, discutiremos as classes e métodos que constituem os pilares da estrutura interna dos códigos desenvolvidos com o Manim. A partir deles, torna-se possível construir animações relacionadas a diversos conteúdos do currículo de matemática, explorando tanto aspectos visuais quanto conceituais. Neste sentido, explicaremos as três categorias fundamentais de componentes: *Mobjects*, *Animations* e *Scenes*. Cada uma dessas categorias desempenhará um papel específico no processo de construção das animações.

3.3.1 Mobjects

Os Mobjects (em referência a *Mathematical Object*) são os elementos visuais fundamentais do Manim. São eles que, de fato, aparecem na tela durante a animação. Eles são modelados pela classe `Mobject`. No entanto, a classe `Mobject` é abstrata, o que significa que não pode ser instanciada diretamente para renderização. Em vez disso, utilizamos especializações da `Mobject`. Por padrão, o Manim vem com diversos módulos que possuem instâncias da `Mobject`, listados a seguir:

- **frame**: retângulos especiais que delimitam ou definem a região visível da cena.
- **geometry**: contém classes para criar formas geométricas básicas (como círculos, quadrados e linhas).
- **graph**: voltado à construção de grafos (do tipo utilizado em teoria dos grafos).

- **graphing**: lida com sistemas de coordenadas e gráficos de funções matemáticas.
- **logo**: fornece utilitários específicos para exibir logotipos e elementos gráficos do próprio Manim.
- **matrix**: destinado à criação e manipulação de matrizes visuais.
- **mobject**: módulo fundamental que contém as classes base para todos os tipos de objetos exibíveis.
- **svg**: permite a importação e manipulação de imagens vetoriais no formato SVG.
- **table**: oferece recursos para a criação de tabelas com estrutura visual.
- **text**: inclui os objetos responsáveis pela exibição de textos, tanto em *LaTeX* quanto via Pango.
- **three_d**: provê suporte para objetos tridimensionais e suas respectivas transformações.
- **types**: contém classes base especializadas para usos específicos de *mobjects*.
- **utils**: utilitários diversos que auxiliam na manipulação de *mobjects*.
- **value_tracker**: objetos simples usados para armazenar e atualizar valores dinamicamente.
- **vector_field**: voltado à representação visual de campos vetoriais.

A seguir, daremos detalhes dos módulos `graphing`, `geometry`, `text` e `mobject`, os quais representam os tipos de objetos visuais mais utilizados para construir nossas animações.

O módulo `geometry`

O módulo `geometry` é subdividido em vários módulos, cada um com um propósito específico na criação de animações geométricas:

- `arc`: Contém Mobjects que representam curvas e arcos.
- `boolean_ops`: Permite a realização de operações booleanas (união, interseção, diferença) entre Mobjects bidimensionais.
- `labeled`: Oferece Mobjects que herdam de linhas e podem conter rótulos ao longo de seu comprimento.
- `line`: Dedicado a Mobjects que são linhas ou variações delas, como segmentos e setas.
- `polygram`: Abriga Mobjects que são formas geométricas simples, porém versáteis, com foco em polígonos e suas generalizações.

- `shape_matchers`: Mobjects utilizados para marcar e anotar outros Mobjects, auxiliando na clareza das apresentações.
- `tips`: Uma coleção de Mobjects de ponta, úteis para a classe `TipableVMobject`.

Dentre os submódulos do módulo `geometry`, o `polygram` será o principal para nossos estudos. Ele nos oferece muitas facilidades ao criar variadas formas geométricas, tornando o processo mais rápido e simplificado. Este módulo contém as seguintes classes:

- `ConvexHull`: Constrói o invólucro convexo de um conjunto de pontos.
- `Cutout`: Cria uma forma com recortes internos.
- `Polygon`: Cria um polígono dados seus vértices.
- `Polygram`: Generaliza `Polygon`, permitindo arestas desconectadas e intersecções.
- `Rectangle`: Cria um retângulo.
- `RegularPolygon`: Polígono regular com "n" lados iguais.
- `RegularPolygram`: `Polygram` com vértices espaçados regularmente.
- `RoundedRectangle`: Retângulo com cantos arredondados.
- `Square`: Cria um quadrado.
- `Star`: Cria uma estrela (um `RegularPolygram` sem linhas de intersecção visíveis).
- `Triangle`: Cria um triângulo equilátero.

A classe `Polygon`. É a classe fundamental para criar qualquer forma poligonal definida por um único laço fechado de vértices. Ela herda funcionalidades da classe `Polygram`, o que lhe confere flexibilidade extra na manipulação. O principal parâmetro para definir um `Polygon` é a variável `vertices`, que aceita uma lista de pontos no formato `[x, y, z]`. A ordem desses vértices é super importante, pois ela dita como as arestas do polígono serão conectadas.

Exemplo de uso.

```
from manim import *

class Poligonos(Scene):
    def construct(self):
```

```
        # Cria um triângulo isósceles definindo explicitamente seus três vértices.
        isosceles = Polygon([-5, 1.5, 0], [-2, 1.5, 0], [-3.5, -2, 0])

        # Define uma lista de posições para criar um polígono mais complexo.
        pontos = [
            [4, 1, 0],
            [4, -2.5, 0],
            [0, -2.5, 0],
            [0, 3, 0],
            [2, 1, 0],
            [4, 3, 0],
        ]
        # Cria um polígono usando a variável 'pontos'.
        poligono = Polygon(*pontos)

        self.add(isosceles, poligono)
```

A classe `Triangle`. É uma especialização que representa um triângulo. Por padrão, ela cria um triângulo equilátero, mas você também pode construí-lo especificando seus vértices, de forma similar ao `Polygon`. É uma forma conveniente quando se precisa de um triângulo sem a necessidade de definir cada vértice manualmente, ou quando se deseja um triângulo perfeito com lados iguais.

Exemplo de uso.

```
from manim import *

class Triangulo(Scene):
    def construct(self):
        #cria o triângulo
        triangulo_1 = Triangle()
        #cria outro triangulo, dobra o tamanho e gira 60º graus anti-horário
        triangulo_2 = Triangle().scale(2).rotate(60*DEGREES)

        self.add(triangulo_1, triangulo_2)
```

A classe `Rectangle`. É usada para criar quadriláteros com lados paralelos, ou seja, retângulos. Seus parâmetros mais comuns são `width` (largura) e `height` (altura).

Exemplo de uso.

```
from manim import *

class ExemploDoisRetangulosSeparados(Scene):
```

```
    def construct(self):
        # Cria o primeiro retângulo.
        retangulo_um = Rectangle(width=4.0, height=2.0)

        # Cria o segundo retângulo.
        retangulo_dois = Rectangle(width=1.0, height=4.0)

        # Adiciona os retângulos à cena.
        self.add(retangulo_um, retangulo_dois)
```

A classe `Square`. É uma particularização de `Rectangle` no Manim, projetada para criar quadrados. Isso significa que ela herda todas as funcionalidades de `Rectangle`, mas com a garantia de que sua largura e altura serão sempre idênticas. Para definir suas dimensões, o parâmetro é `side_length` (comprimento do lado). Optar por `Square` simplifica a criação de elementos com proporção 1:1.

Exemplo de uso.

```
from manim import *

class Quadrado(Scene):
    def construct(self):
        # Cria  quadrado com lado de comprimento 2
        quadrado_um = Square(side_length=2.0)

        # Cria o segundo quadrado com lado de comprimento 1
        quadrado_dois = Square(side_length=1.0)

        # Adiciona os dois quadrados à cena.
        self.add(quadrado_um, quadrado_dois)
```

A classe `Circle`. Embora não pertença diretamente à ramificação `polygram`, a classe `Circle` é um Mobject geométrico fundamental no Manim. Ela reside no módulo `arc`, do módulo geometry, que se dedica a formas curvas. A criação de um círculo é bastante intuitiva, sendo comumente definida pelo o raio.

Exemplo de uso.

```
from manim import *

class Circulo(Scene):
    def construct(self):
```

```
        # Cria um círculo com raio de 2 unidades
        circulo = Circle(radius=2)

        # Adiciona o círculo à cena.
        self.add(circulo)
```

O módulo `graphing`

O módulo `graphing` é estruturado em outros submódulos, cada um com foco em funcionalidades específicas para a representação gráfica. Essa organização permite uma abordagem modular na construção de visualizações de dados e funções, tornando mais acessível e simples as construções de funções e gráficos. Veja abaixo as divisões desse módulo:

- **coordinate_systems**
  - `Axes` – Cria um conjunto de eixos coordenados.
  - `ComplexPlane` – Um plano cartesiano para números complexos.
  - `CoordinateSystem` – Classe base abstrata para `Axes` e `NumberPlane`.
  - `NumberPlane` – Cria um plano cartesiano com linhas de grade de fundo.
  - `PolarPlane` – Cria um plano polar com linhas de grade de fundo.
  - `ThreeDAxes` – Eixos coordenados em 3 dimensões.
- **functions**
  - `FunctionGraph` – Uma função paramétrica que se estende por todo o comprimento da cena por padrão.
  - `ImplicitFunction` – Uma função implícita.
  - `ParametricFunction` – Uma curva paramétrica.
- **number_line**
  - `NumberLine` – Cria uma reta numérica com marcas de escala.
  - `UnitInterval` – Reta numérica representando o intervalo $[0, 1]$.
- **probability**
  - `BarChart` – Cria um gráfico de barras.
  - `SampleSpace` – Um mobject que representa um espaço amostral retangular bidimensional.
- **scale**

– `LinearBase` – Classe padrão para escalas lineares.

– `LogBase` – Escala para gráficos/funções logarítmicas.

No nosso estudo, não houve necessidade de usar todas essas opções listadas, as que mais usamos e que foram fundamentais nas nossas animações foram:

- `Axes`
- `NumberPlane`
- `FunctionGraph`
- `NumberLine`

A seguir, cada uma das classes citadas acima é apresentada com uma descrição de seu uso e um exemplo simples. O funcionamento das outras classes pode ser estudado consultando o site da Comunidade oficial do Manim.

`Axes`. A classe `Axes` permite criar eixos coordenados no plano cartesiano, com controle sobre escala, rótulos e estilo dos eixos. Ela é usada como base para plotar funções e destacar pontos no plano.

Exemplo de uso.

```
from manim import *

class EixosSimples(Scene):
    def construct(self):
        eixos = Axes(
            x_range=[-3, 3],
            y_range=[-2, 2],
            tips=False
        )
        #tips=false é para não mostrar setas nas pontas dos eixos
        self.play(Create(eixos))
```

`NumberPlane`. A classe `NumberPlane` herda de `Axes` e adiciona linhas de grade ao plano, tornando mais fácil visualizar posições relativas e simetrias. É ideal para cenas que exploram o plano cartesiano completo.

Exemplo de uso.

```
from manim import *

class PlanoComGrade(Scene):
    def construct(self):
        plano = NumberPlane()
        self.play(Create(plano))
```

`FunctionGraph`. A classe `FunctionGraph` permite desenhar o gráfico de uma função usando um eixo definido previamente (como o criado com `Axes`). É bastante flexível e permite destacar comportamentos de funções matemáticas.

Exemplo de uso.

```
from manim import *

class GraficoDeFuncao(Scene):
    def construct(self):
        eixos = Axes(
            x_range=[-3, 3, 1],
            y_range=[-2, 8, 1],
            tips=False
        )
        parabola = eixos.plot(lambda x: x**2, color=BLUE)
        self.play(Create(eixos), Create(parabola))
```

`NumberLine`. A classe `NumberLine` é usada para representar retas numéricas, com marcas e rótulos. É útil em situações didáticas que envolvem números reais, inteiros, intervalos ou localização de pontos.

Exemplo de uso.

```
from manim import *

class RetaNumerica(Scene):
    def construct(self):
        retanumerica = NumberLine(x_range=[-4, 4], include_numbers=True)
        self.play(Create(retanumerica))
```

O módulo `text`

O módulo `text` é responsável por fornecer as ferramentas necessárias para exibir textos nas animações. Ele permite utilizar diferentes motores de renderização, como `Pango` (texto simples com suporte a Unicode) e LaTeX (para fórmulas matemáticas e símbolos científicos). Essa flexibilidade é essencial para realização das animações, pois possibilita a incorporação de conteúdos matemáticos formais.

A estrutura interna desse módulo é a seguinte:

- **text:**
  - `Code` - Exibe trechos de código com realce de sintaxe.
- **numbers:**

  - `DecimalNumber` - Representa números decimais formatáveis.
  - `Integer` - Representa inteiros como texto.
  - `Variable` - Combina texto com valor numérico exibido.

- **tex_mobject:**
  - `BulletedList` - Cria listas com marcadores usando LaTeX.
  - `MathTex` - Renderiza expressões LaTeX em modo matemático.
  - `SingleStringMathTex` - Variante simples de MathTex para uma única string.
  - `Tex` - Renderiza texto LaTeX em modo normal (não matemático).
  - `Title` - Mobject para criar títulos sublinhados com LaTeX.

- **text_mobject:**
  - `MarkupText` - Usa marcação estilo HTML (ex: <b>, <i>) para estilizar texto.
  - `Paragraph` - Organiza múltiplas linhas de texto como parágrafos.
  - `Text` - Exibe texto simples utilizando o motor Pango.

No contexto desta dissertação, os elementos do módulo `text` mais utilizados foram:

- `Text`
- `MathTex`
- `Paragraph`

As demais classes, embora úteis em contextos específicos, não foram essenciais para as animações desenvolvidas neste trabalho. A seguir, apresentamos as classes utilizadas, com exemplos simples de implementação.

`Text.` A classe `Text` permite a exibição de texto simples na tela. Seu uso é ideal para títulos, legendas ou qualquer informação textual que não dependa de simbologia matemática.

Exemplo de uso.

```
from manim import *

class TextoSimples(Scene):
    def construct(self):
        texto = Text("Olá, mundo!", font_size=48)
        self.play(Write(texto))
```

`MathTex`. A classe `MathTex` é utilizada para escrever fórmulas e expressões matemáticas com o compilador LaTeX. Isso é essencial para visualizações em contextos acadêmicos.

Exemplo de uso.

```
from manim import *

class Formula(Scene):
    def construct(self):
        formula = MathTex("a^2 = b^2 + c^2")
        self.play(Write(formula))
```

`Paragraph`. A classe `Paragraph` é usada para apresentar blocos de texto com múltiplas linhas, útil para explicações mais longas ou instruções dentro da cena.

Exemplo de uso.

```
from manim import *

class Paragrafo(Scene):
    def construct(self):
        paragrafo = Paragraph(
            r"Olá! A matemática é bela!", # Primeira linha de texto
            r"Funções descrevem o mundo", # Segunda linha de texto
            alignment="left", # Alinha o texto à esquerda
            line_spacing=0.8, # Define o espaçamento entre linhas
            font="CMU Serif" # Define a fonte do texto
        ).scale(0.35) # Reduz o tamanho do parágrafo
        self.play(Write(paragrafo))
```

O módulo `mobject`

O módulo `mobject` é a base para animação dos projetos desenvolvidos com Manim. Ele define a estrutura e os comportamentos dos objetos gráficos utilizados nas cenas. Esses elementos são fundamentais para compor animações, abrangendo desde textos e formas geométricas até gráficos e construções matemáticas mais complexas

A estrutura interna desse módulo é a seguinte:

- `Mobject` – Classe base para todos os objetos gráficos no Manim. Define métodos fundamentais como animação, posicionamento e transformação.
- `Group` – Permite agrupar múltiplos Mobjects em uma única unidade lógica e visual, facilitando animações e manipulações coletivas.

`Mobject`. A classe `Mobject` é a classe base para todos os elementos visuais em Manim. Embora geralmente não seja utilizada diretamente, ela define os métodos fundamentais que todas as outras classes herdadas utilizam. Com `Mobject`, é possível aplicar transformações como escala, translação, rotação, além de controlar propriedades visuais como cor, opacidade e posição.

Os métodos mais utilizados desta classe incluem:

- `animate`: transforma operações estáticas em animações dinâmicas.
- `shift()`, `move_to()`, `next_to()`: para movimentar objetos.
- `set_color()`, `get_color()`: para alterar ou consultar cores.
- `scale()`, `rotate()`: para transformar a forma visual do objeto.
- `add_updater()`: permite atualizações contínuas nos objetos.
- `add()`, `remove()`: para gerenciamento de submobjects.

Exemplo de uso.

```
from manim import *

class ExemploMobject(Scene):
    def construct(self):
        circulo = Circle(radius=1, color=BLUE)
        self.play(circulo.animate.shift(RIGHT * 2).scale(1.5).set_color(RED))
```

Nesse exemplo, o `Circle` herda de `Mobject`, e o método `animate` transforma as operações encadeadas (`shift`, `scale`, `set_color`) em uma animação bem mais elegante.

Abaixo, apresentamos o código para ilustrar os demais conceito.

```
from manim import *

class Exemplo(Scene):
    def construct(self):
        # Cria um quadrado azul e move duas unidades para a esquerda
        quadrado = Square(color=BLUE).shift(LEFT * 2)

        # Cria um triângulo vermelho e coloca acima do quadrado
        triangulo = Triangle(color=RED).next_to(quadrado, UP)

        # Adiciona um "updater" ao quadrado
        # O updater fará o quadrado girar continuamente
        #"dt" duração do frame anterior. Garante que seja suave a animação.
        quadrado.add_updater(lambda m, dt: m.rotate(PI/4 * dt))
```

```
        # Adiciona um "updater" ao triângulo
        # Ele sempre se posicionará acima do quadrado
        triangulo.add_updater(lambda m: m.next_to(quadrado, UP))

        # Adiciona os mobjects à cena
        self.add(quadrado, triangulo)

        # Espera 5 segundos para a animação ocorrer
        self.wait(5)

        # Remove os updaters para parar a animação contínua
        quadrado.remove_updater(lambda m, dt: m.rotate(PI/4 * dt))
        triangulo.remove_updater(lambda m: m.next_to(quadrado, UP))

        # Exemplo de uso de animate.rotate para uma rotação única
        self.play(quadrado.animate.rotate(PI/2))
```

Group. A classe `Group` permite agrupar múltiplos Mobjects em uma única estrutura. Isso é especialmente útil quando se deseja aplicar animações ou transformações a vários objetos simultaneamente, como mover um título, subtítulo e figura juntos.

Exemplo de uso.

```
from manim import *

class Agrupamento(Scene):
    def construct(self):
        texto1 = Text("Título")
        texto2 = Text("Subtítulo").next_to(texto1, DOWN)
        grupo = Group(texto1, texto2)
        self.play(FadeIn(grupo))
        self.play(grupo.animate.shift(UP * 2))
```

Neste exemplo, os dois textos são agrupados com `Group`, permitindo que sejam animados conjuntamente com apenas uma chamada de método.

Atributos das classes

As classes dos modulos `geometry`, `graphing` e `mobjects` compartilham uma série de atributos visuais e comportamentais que podem ser ajustados para compor a cena. Alguns já foram exploradas nos códigos apresentados anteriormente, mas há outros atributos que podemos usar para melhorar a qualidade das animações.

De forma geral, os principais atributos disponíveis são:

- `color`: define a cor da borda do objeto. Pode-se utilizar nomes como `BLUE`, `RED`, `GREEN`, entre outros.
- `fill_color`: determina a cor de preenchimento interno da forma.
- `fill_opacity`: controla a opacidade do preenchimento, variando de `0` (totalmente transparente) a `1` (totalmente opaco).
- `stroke_color`: cor do contorno (semelhante a `color`, mas pode ser configurada independentemente).
- `stroke_width`: espessura da linha de contorno.
- `sheen` e `sheen_direction`: criam efeitos de brilho direcional sobre o preenchimento do objeto.
- `shade_in_3d`: permite simular sombreamento 3D ao combinar com iluminação na cena tridimensional.
- `glow_color`: adiciona um efeito de brilho ao redor do objeto.
- `rotate`: método que rotaciona o objeto.

Eles podem ser passados diretamente na criação do objeto ou ajustados posteriormente com métodos como `.set_color()`, `.set_fill()`, `.set_stroke()`, entre outros.

Exemplo de uso.

```
from manim import *

class Quadrado(Scene):
    def construct(self):
        # Cria um quadrado com lado de comprimento 2,
        # preenchido com cor azul e contorno branco.
        quadrado_um = Square(
            side_length=2.0,      # Comprimento do lado
            color=WHITE,          # Cor do contorno
            fill_color=BLUE,      # Cor de preenchimento
            fill_opacity=0.5,     # Transparência do preenchimento (0 a 1)
            stroke_width=4        # Espessura do contorno
        )

        # Cria o segundo quadrado com lado menor,
        # borda vermelha, preenchido em amarelo opaco e rotacionado
        quadrado_dois = Square(
            side_length=1.0,
            color=RED,
            fill_color=YELLOW,
            fill_opacity=0.8,
            stroke_width=2
```

```
).rotate(PI/4)

# Adiciona os dois quadrados à cena.
self.add(quadrado_um, quadrado_dois)
```

3.3.2 Animations

As *Animations* são responsáveis por dar vida aos Mobjects. A classe `Animation` também é abstrata, sendo utilizada de forma indireta por meio de suas subclasses, como `Create`, `Write`, `FadeIn`, `FadeOut`, `Rotate` e `MoveToTarget`.

Essas animações permitem adicionar elementos visualmente à cena, modificar suas posições ou aparências, e criar transições visuais significativas. Algumas delas, como `ReplacementTransform` e `ApplyMethod`, são úteis para substituições ou transformações mais complexas.

A organização da classe `Animation` é bastante simples, mas há muitos comandos, agrupados em diferentes categorias, conforme a funcionalidade que oferecem. Abaixo, apresentamos a classificação completa da classe `Animations` na biblioteca Manim, acompanhados de uma breve descrição de cada comando.

- **animation** – Comandos básicos de animação:
  - `Add`, `Animation`, `Wait`
- **changing** – Alteram dinamicamente objetos ao longo do tempo:
  - `AnimatedBoundary`, `TracedPath`
- **composition** – Permitem organizar animações em grupo:
  - `AnimationGroup`, `LaggedStart`, `LaggedStartMap`, `Succession`
- **creation** – Criam ou revelam objetos:
  - `AddTextLetterByLetter`, `AddTextWordByWord`, `Create`, `DrawBorderThenFill`, `RemoveTextLetterByLetter`, `ShowIncreasingSubsets`, `ShowPartial`, `ShowSubmobjectsOneByOne`, `SpiralIn`, `TypeWithCursor`, `Uncreate`, `UntypeWithCursor`, `Unwrite`, `Write`
- **fading** – Efeitos de desaparecimento e surgimento:
  - `FadeIn`, `FadeOut`
- **growing** – Apresentam objetos com efeito de crescimento:
  - `GrowArrow`, `GrowFromCenter`, `GrowFromEdge`, `GrowFromPoint`, `SpinInFromNothing`

- **indication** – Destacam visualmente objetos:
  - `ApplyWave`, `Blink`, `Circumscribe`, `Flash`, `FocusOn`, `Indicate`, `ShowPassingFlash`, `ShowPassingFlashWithThinningStrokeWidth`, `Wiggle`
- **movement** – Movimentações específicas e contínuas:
  - `ComplexHomotopy`, `Homotopy`, `MoveAlongPath`, `PhaseFlow`, `SmoothedVectorizedHomotopy`
- **numbers** – Atualização dinâmica de valores:
  - `ChangeDecimalToValue`, `ChangingDecimal`
- **rotation** – Animações que envolvem rotação:
  - `Rotate`, `Rotating`
- **specialized** – Efeitos únicos e específicos:
  - `Broadcast`
- **speedmodifier** – Alteram a velocidade de animações:
  - `ChangeSpeed`
- **transform** – Transformam objetos em outros:
  - `ApplyComplexFunction`, `ApplyFunction`, `ApplyMatrix`, `ApplyMethod`, `ApplyPointwiseFunction`, `ApplyPointwiseFunctionToCenter`, `ClockwiseTransform`, `CounterclockwiseTransform`, `CyclicReplace`, `FadeToColor`, `FadeTransform`, `FadeTransformPieces`, `MoveToTarget`, `ReplacementTransform`,
    `Restore`, `ScaleInPlace`, `ShrinkToCenter`, `Swap`,
    `Transform`, `TransformAnimations`, `TransformFromCopy`
- **transform_matching_parts** – Transformações que respeitam partes correspondentes:
  - `TransformMatchingAbstractBase`, `TransformMatchingShapes`, `TransformMatchingTex`
- **updaters** – Atualizações ligadas a funções e tempo:
  - `mobject_update_utils`, `update`

Nos projetos desta dissertação, destacam-se algumas animações essenciais e frequentemente utilizadas para apresentar conceitos matemáticos. A seguir, discutimos as animações mais recorrentes, com exemplos representativos de uso.

- `Create`: traça a borda de um objeto na tela, como se estivesse sendo desenhado. É muito utilizada para destacar contornos de figuras ou gráficos.

```
class CriarObjeto(Scene):
    def construct(self):
        circulo = Circle()
        self.play(Create(circulo))
```

- `Write`: similar ao `Create`, mas ideal para texto, simulando a escrita das letras.

```
class EscreverTexto(Scene):
    def construct(self):
        texto = Text("Função exponencial")
        self.play(Write(texto))
```

- `FadeIn` e `FadeOut`: permitem o aparecimento ou desaparecimento suave dos objetos na cena.

```
class Desvanecer(Scene):
    def construct(self):
        quadrado = Square()
        self.play(FadeIn(quadrado))
        self.wait(1)
        self.play(FadeOut(quadrado))
```

- `Rotate`: realiza a rotação de um objeto em torno do seu centro ou de um ponto específico.

```
class Rotacionar(Scene):
    def construct(self):
        tri = Triangle()
        self.play(Rotate(tri, angle=PI/2))
```

- `ReplacementTransform`: substitui suavemente um objeto por outro, mantendo coerência visual.

```
class Transformar(Scene):
    def construct(self):
        c1 = Circle()
        c2 = Square()
        self.play(ReplacementTransform(c1, c2))
```

- `MoveToTarget`: movimenta um objeto até uma posição alvo definida previamente por `generate_target()`.

```
class MoverParaAlvo(Scene):
    def construct(self):
        txt = Text("Movendo")
        txt.generate_target()
        txt.target.to_edge(UP)
        self.add(txt)
        self.play(MoveToTarget(txt))
```

- `ChangeDecimalToValue`: usada para animar a transição numérica em um `DecimalNumber`.

```
class MudarValor(Scene):
    def construct(self):
        numero = DecimalNumber(0)
        self.add(numero)
        self.play(ChangeDecimalToValue(numero, 5))
```

É possível combinar diversas animações em um único código, desde que sejam organizadas de forma lógica. O design da animação depende não apenas da escolha dos métodos, mas também da maneira como são encadeados para comunicar visualmente uma ideia. A seguir, apresenta-se um exemplo que reúne várias das animações discutidas, demonstrando sua aplicação conjunta em um único projeto.

```
from manim import *

class ProjetoAnimacoes(Scene):
    def construct(self):
        # Texto inicial
        titulo = Text("Explorando Animações", font_size=48)
        self.play(Write(titulo))
        self.wait(1)
        self.play(FadeOut(titulo))

        # Criação de formas geométricas
        circ = Circle().shift(LEFT*2)
        quad = Square().shift(RIGHT*2)
        self.play(Create(circ), Create(quad))

        # Transição entre formas
        self.wait(1)
        self.play(ReplacementTransform(circ, quad))
```

```
        # Adição de uma seta com crescimento
        seta = Arrow(LEFT, RIGHT, buff=0.2).shift(DOWN*2)
        self.play(GrowArrow(seta))

        # Texto com número dinâmico
        numero = DecimalNumber(0).to_edge(UP)
        self.add(numero)
        self.play(ChangeDecimalToValue(numero, 100, run_time=2))

        # Movimento de um objeto ao longo de um caminho
        caminho = Arc(radius=2, angle=PI).shift(DOWN)
        ponto = Dot().move_to(caminho.point_from_proportion(0))
        self.add(ponto)
        self.play(MoveAlongPath(ponto, caminho), run_time=3)

        # Aplicando rotação
        rotulo = Text("Rotacionando", font_size=32).next_to(quad, DOWN)
        self.play(Write(rotulo))
        self.play(Rotate(quad, angle=PI))

        # Destaque com circunscrição
        self.play(Circumscribe(rotulo, color=YELLOW))

        # Desaparecendo com FadeOut
        self.play(FadeOut(rotulo), FadeOut(quad), FadeOut(seta),
            FadeOut(ponto), FadeOut(numero))

        # Texto final
        fim = Text("Fim da apresentação!", font_size=44)
        self.play(TypeWithCursor(fim))
        self.wait(2)
```

3.3.3 Scenes

A classe `Scene` é o núcleo de qualquer animação no Manim. Toda animação precisa estender uma cena. No entanto, a biblioteca oferece várias subclasses especializadas de `Scene` que permitem interações mais sofisticadas, como câmeras móveis, transformações lineares, cenas 3D, entre outras. Apesar de sua importância, este nível de abstração é mais avançado e será melhor explorado por aqueles que desejam se aprofundar na estrutura do Manim ou desenvolver animações altamente persona-

lizadas. Aqui daremos uma breve visão geral, apenas como referência. A seguir, apresentamos um panorama geral dessas subclasses com exemplos ilustrativos.

- **moving_camera_scene**
  - `MovingCameraScene` – Permite que a câmera se mova e se aproxime dos objetos. Ideal para destacar partes específicas de uma cena.

```
from manim import *

class MoveCamExample(MovingCameraScene):
    def construct(self):
        square = Square()
        self.add(square)
        self.camera.frame.save_state()
        self.play(
            self.camera.frame.animate.set(width=2).move_to(square))
        self.wait()
```

- **section**
  - `DefaultSectionType` – Permite categorizar seções de uma cena para controle avançado.
  - `Section` – Define pontos de parada, marcações ou categorias em uma cena longa, útil em apresentações interativas.

```
class SectionExample(Scene):
    def construct(self):
        self.next_section("Intro",
            type=DefaultSectionType.SUBSCENE)
        self.play(Write(Text("Introdução")))
        self.next_section("Conclusão")
        self.play(Write(Text("Fim")))
```

- **scene**
  - `RerunSceneHandler` – Detecta alterações no código-fonte e recompila a cena automaticamente.
  - `Scene` – Classe base de qualquer animação. Todas as outras derivam dela. Contém os métodos fundamentais de animação.

```
class BasicScene(Scene):
    def construct(self):
        self.play(Create(Circle()))
        self.wait()
```

- **scene_file_writer**
  - `SceneFileWriter` – Controla a gravação, renderização e exportação das cenas. Ideal para ajustes personalizados de saída.

```
from manim.scene.scene_file_writer import SceneFileWriter

writer = SceneFileWriter(scene=None, file_name="teste")
writer.begin_animation()
writer.end_animation()
```

- **three_d_scene**
  - `SpecialThreeDScene` – Extensão de ThreeDScene com mais opções.
  - `ThreeDScene` – Ideal para animações com profundidade. Permite rotacionar e posicionar objetos 3D.

```
class ThreeDExample(ThreeDScene):
    def construct(self):
        axes = ThreeDAxes()
        sphere = Sphere()
        self.set_camera_orientation(phi=75 * DEGREES, theta=30 * DEGREES)
        self.add(axes, sphere)
        self.wait()
```

- **vector_space_scene**
  - `LinearTransformationScene` – Foca em transformações lineares visuais. Ideal para representar multiplicações de vetores por matrizes.
  - `VectorScene` – Fornece ferramentas para desenhar e manipular vetores.

```
class LinearTransformExample(LinearTransformationScene):
    def __init__(self, **kwargs):
        super().__init__(
            show_coordinates=True,
            show_basis_vectors=True,
            **kwargs
        )

    def construct(self):
        matrix = [[1, 1], [0, 1]]
        self.apply_matrix(matrix)
```

- **zoomed_scene**
  - `ZoomedScene` – Permite criar janelas de zoom para destacar partes específicas da cena.

```
class ZoomExample(ZoomedScene):
    def construct(self):
        dot = Dot().shift(UP)
        self.add(dot)
        self.activate_zooming()
        self.wait()
```

3.4 Vídeos narrados

A produção de vídeos animados com narração, à primeira vista, pode revelar-se uma tarefa complexa, sobretudo devido às etapas de gravação e edição manual do áudio. No entanto, a biblioteca `Manim` dispõe de uma solução eficiente por meio da extensão `manim-voiceover`, a qual permite incorporar narração diretamente ao código da animação, utilizando serviços de conversão de texto em fala (TTS, do inglês *Text-to-Speech*).

Para utilizar este recurso, é necessário instalar um pacote adicional com o seguinte comando:

```
pip install "manim-voiceover[azure,gtts]"
```

Esse comando instala os serviços de voz do Google (`gTTS`) e da Microsoft Azure. O `gTTS` é gratuito, mas gera vozes com aspecto mais artificial, sendo útil para testes rápidos ou protótipos. Já o serviço da `Azure`, embora mais realista e natural, requer uma conta na plataforma e está sujeito a cobrança, especialmente para produções maiores.

O módulo gTTS

Com o pacote instalado, iniciamos importando os módulos necessários:

```
from manim_voiceover import VoiceoverScene
from manim_voiceover.services.gtts import GTTSService
```

Em seguida, definimos a cena como sendo do tipo `VoiceoverScene`, podendo combiná-la com outras, como `MovingCameraScene` ou `ThreeDScene`. Depois, configuramos o serviço de voz a ser usado, como no exemplo abaixo:

```
self.set_speech_service(GTTSService(lang="pt"))
```

Exemplo de uso.

O trecho a seguir mostra como adicionar uma narração que sincroniza a fala com a animação:

```
from manim import *
from manim_voiceover import VoiceoverScene
from manim_voiceover.services.gtts import GTTSService

class AudioGTTSExample(VoiceoverScene):
    def construct(self):
        self.set_speech_service(GTTSService(lang="pt"))
        circle = Circle()

        with self.voiceover(
        text="Estou desenhando um círculo enquanto falo"
        ) as tracker:
            self.play(Create(circle), run_time=tracker.duration)
```

Para que a duração da animação seja adaptada ao tempo da narração, utilizamos o tempo estimado pela variável `tracker.duration`, como usado acima.

O módulo Azure

Para utilizar o serviço de voz do `Azure`, que oferece vozes mais naturais e realistas, basta importar o serviço e configurar da mesma forma:

```
from manim_voiceover.services.azure import AzureService
```

Exemplo de uso.

```
from manim import *
from manim_voiceover import VoiceoverScene
from manim_voiceover.services.azure import AzureService

class AudioAzureExample(VoiceoverScene):
    def construct(self):
        self.set_speech_service(
            AzureService(voice="pt-BR-FranciscaNeural")
        )
        square = Square()

        with self.voiceover(
        text="Este é um quadrado com narração realista"
        ) as tracker:
            self.play(Create(square), run_time=tracker.duration)
```

É importante observar que o uso da API da Azure pode incorrer em custos. A Microsoft oferece uma camada gratuita limitada, mas para produções extensas será necessário contratar um plano. A documentação oficial contém exemplos de vozes disponíveis, permitindo ao usuário escolher o idioma, sotaque e entonação.

Usando a própria voz

Para aqueles que preferem gravar suas próprias falas, o `manim-voiceover` também oferece suporte. Basta importar:

```
from manim_voiceover.services.recorder import RecorderService
```

A seguir, um exemplo completo com gravação manual:

```
from manim import *
from manim_voiceover import VoiceoverScene
from manim_voiceover.services.recorder import RecorderService

class AudioGravadoExample(VoiceoverScene):
    def construct(self):
        self.set_speech_service(RecorderService())
        triangle = Triangle()

        with self.voiceover(
        text="Agora estou desenhando um triângulo"
        ) as tracker:
            self.play(Create(triangle), run_time=tracker.duration)
```

Durante a renderização, o sistema solicitará a gravação manual da narração, executando as animações no tempo determinado pelo código. Instruções na linha de comando orientarão sobre o momento da narração, permitindo ao usuário acompanhar a cena ao vivo e gravar sua voz em sincronia com a animação.

Capítulo 4

SAEPE Animado

Com o intuito de explorar o uso da biblioteca `Manim` como ferramenta didática para animar conteúdos matemáticos presentes nas avaliações externas, com foco específico no SAEPE, elaboramos vídeos animados que abordam os quatro eixos temáticos dessa avaliação, a saber, Geometria, Grandezas e Medidas, Números e Operações/Álgebra de Funções e Estatística, Probabilidade e Combinatória. A ideia é articular o conteúdo matemático à sua visualização dinâmica, permitindo ao estudante compreender com maior clareza a lógica por trás das fórmulas e da resolução de problemas, além disso, incentivando o desenvolvimento do pensamento computacional.

A seguir, apresentamos dois dos códigos desenvolvidos com a biblioteca Manim que introduzem visualmente os conceitos de Progressão Aritmética e Área do Trapézio. Para cada um desses temas, foi elaborada uma animação específica que aborda inicialmente as definições e fórmulas matemáticas envolvidas e depois os explora de forma dinâmica e comentada na resolução de uma questão representativa extraída do SAEPE. A proposta busca integrar linguagem visual, interpretação contextual e resolução algébrica, facilitando a compreensão dos conceitos exigidos em avaliações externas. Todas as animações produzidas estão disponíveis em:

`https://onmat.github.io/SAEPE-Animado/`.

Esse ambiente virtual serve como repositório aberto dos materiais produzidos, permitindo o acesso às animações, códigos-fonte comentados e questões complementares, com o objetivo de ampliar as possibilidades de estudo.

4.1 Abertura

Cada animação inicia com uma abertura, cujo código está listado a seguir. Ela começa com a exibição do título "SAEPE Animado", que se desloca para cima e abre espaço para o surgimento de quatro polígonos coloridos alinhados, um triângulo, um

retângulo, um pentágono e um hexágono. Esse deslocamento do texto é realizado pela classe `Transform`. Ela tem como parâmetros as posições inicial e final dos objetos que serão animados, além do caminho que será usado para o deslocamento e o tempo que deve ser gasto durante a movimentação. Além disso, a função `shift` é usada tanto para transladar os polígonos quanto para obter a posição final desejada do título.

Neste início de código, são definidos quatro objetos geométricos utilizando as classes `Triangle`, `Rectangle` e `RegularPolygon`, cada um com cor e dimensões específicas. Em seguida, as figuras são posicionadas horizontalmente na tela por meio do método `.shift()`, evitando que fiquem sobrepostas e garantindo uma distribuição visualmente equilibrada. A animação `Create()` é aplicada simultaneamente a todos os elementos, produzindo um efeito de desenho gradual das formas. Por fim, o comando `self.wait(1)` mantém a cena estática por um segundo. A imagem a seguir retrata de maneira estátitica essa parte do vídeo.

```
from manim import *
from pathlib import Path
import os

class cena(Scene):
    def construct(self):
        # ABERTURA DO VÍDEO
        inicio = Text("SAEPE Animado")
        fim = inicio.copy()
        triangulo = Triangle(color = BLUE)
        retangulo = Rectangle(width=2.5, height=1.5, color=GREEN)
        pentagono = RegularPolygon(n=5, radius=1, color=ORANGE)
        hexagono = RegularPolygon(n=6, radius=1, color=PURPLE)

        self.wait()
        self.play(Write(inicio))
        self.wait()
        self.play(Transform(
                inicio,
                fim.shift(2*UP),
                path_func=utils.paths.straight_path(),
                run_time=2,
            ))
        self.wait()

        triangulo.shift(LEFT * 4.5)
```

```
retangulo.shift(LEFT * 1.6)
pentagono.shift(RIGHT * 1.6)
hexagono.shift(RIGHT * 4.5)

self.play(Create(triangulo), Create(retangulo),
          Create(pentagono), Create(hexagono))
self.wait(1)

# AGRUPAMENTO E REDUÇÃO DAS FIGURAS
grupo1 = VGroup(triangulo, retangulo, pentagono, hexagono)
self.play(grupo1.animate.scale(0.7))
self.wait(0.5)

for _ in range(4):
    self.play(CyclicReplace(*grupo1))
self.wait(1)
```

Neste trecho, as quatro figuras criadas anteriormente são agrupadas por meio da classe `VGroup`, permitindo manipular todas as figuras ao mesmo tempo. O grupo é transformado utilizando `.scale(0.7)` e, em seguida, aplica-se o método `CyclicReplace` em um laço para alternar ciclicamente as posições das figuras. O frame final da abertura está ilustrado na Figura 4.1.

Figura 4.1: Frame final da abertura das animações.

Fonte: Elaborado pelo autor.

4.2 Identificação do tema

Após a abertura, o vídeo é identificado com o eixo temático da matriz de referência e o assunto abordado. Para tanto, é criado um elemento com a classe `Text` para o eixo, posicionado abaixo do `grupo1`, definido na abertura. O posicionamento é realizado

por meio do método `.next_to()`. Todos esses elementos são então agrupados em um `VGroup` para serem removidos posteriormente com o `FadeOut`.

```
# Agora adiciona o texto e agrupa tudo
titulo_gp = Text(
"NÚMEROS E OPERAÇÕES/ÁLGEBRA E FUNÇÕES",
    font="Fira Sans").scale(0.75)
titulo_gp.next_to(grupo1, DOWN, buff=0.8)

self.play(Write(titulo_gp))
self.wait(1)

grupo_com_texto = VGroup(grupo1, titulo_gp, inicio)
self.play(FadeOut(grupo_com_texto))
```

Na sequência, o assunto abordado é exibido sublinhado usando a classe `Text`, acompanhado de uma linha (`Line`) posicionada logo abaixo. Essa linha recebe espessura e cor definidas, e um *updater* (`.add_updater`) que altera dinamicamente sua cor utilizando a função `interpolate_color` combinada com `np.sin`, gerando uma oscilação entre vermelho e azul. Por fim, a linha e o texto são agrupados e reposicionados no canto superior esquerdo da tela, preservando sua proporção por meio de `.scale` e `.to_corner`.

```
t1 = Text("Progressão Aritmética", font="Fira Sans").scale(0.5)
self.play(Write(t1, run_time = 2))
self.wait(0.5)

linha = Line(
    start=t1.get_bottom() + DOWN * 0.2 + LEFT * t1.width / 2,
    end=t1.get_bottom() + DOWN * 0.2 + RIGHT * t1.width / 2,
    stroke_width=6,
)
linha.set_color(RED)

# Função para atualizar a cor da linha
def atualizar_linha(obj, dt):
    t = self.time
    nova_cor = interpolate_color(RED, BLUE, (np.sin(t * 2) + 1) / 2)
    obj.set_color(nova_cor)

linha.add_updater(atualizar_linha)
self.play(FadeIn(linha))
```

Figura 4.3: Dois animais.

```
self.wait(1)

grupo2 = VGroup(linha, t1)
self.play(grupo2.animate.scale(0.9).to_corner(UP + LEFT), run_time=1.5)
self.wait(1.5)
```

Na Figura 4.2, são apresentados os frames correspondentes à exibição do eixo temático e do assunto de uma animação sobre progressão aritmética.

Figura 4.2: Exibição do eixo temático e do assunto abordados em uma animação.

(a) Eixo temático.

(b) Assunto abordado.

Fonte: Elaborado pelo autor.

4.3 Progressão aritmética

De acordo com a matriz de referência do SAEPE para o 3º ano do Ensino Médio, a habilidade D21 estabelece que o estudante seja capaz de resolver problema envolvendo progressões aritméticas e geométricas dada a fórmula do termo geral. Essa competência exige não apenas o conhecimento da fórmula algébrica, mas também a compreensão de seu significado dentro de contextos práticos e aplicados.

4.3.1 Fundamentação

A apresentação da teoria sobre progressão aritmética consiste de duas etapas: conceituação e dedução.

Na primeira parte da animação, constrói-se uma sequência visual para introduzir a ideia de progressão aritmética. Um `Paragraph` apresenta a definição de forma textual e alinhada à esquerda com o método `left`, seguido pela criação de quadrados que representam graficamente a sequência, dispostos e rotulados de acordo com seus termos. O código explora a função `align_to` para manter a consistência visual, utilizando deslocamentos para posicionar cada elemento. Em seguida, outro `Paragraph` explica a razão da progressão e um terceiro indica a formulação de uma função, que é exibida junto a exemplos específicos com o `MathTex`. O uso de `FadeOut` no final limpa a cena de forma gradual, ao invés de usar o `clear`, que eliminaria todos os objetos de uma vez, mas sem nenhum efeito.

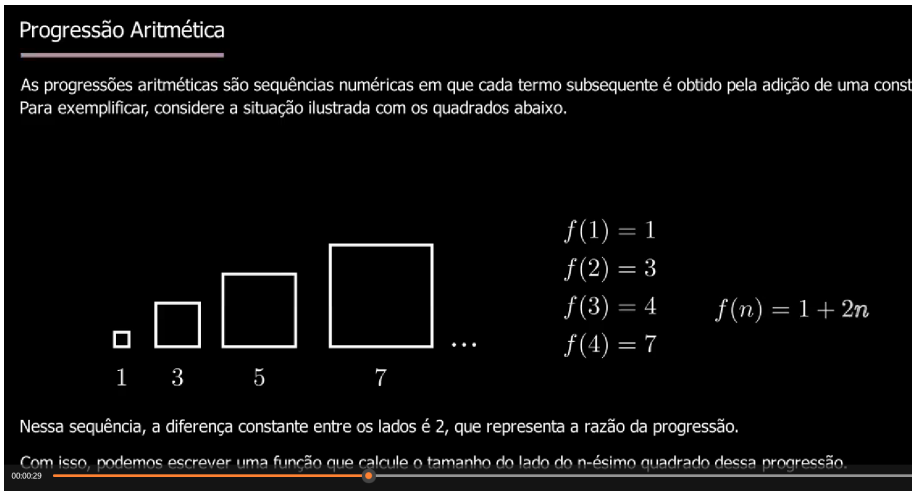
```
explicacao = Paragraph(
    r"As progressões aritméticas são sequências numéricas em que cada"
    r"termo subsequente é obtido pela adição de uma constante.",
    r"Para exemplificar, considere a situação ilustrada com os quadrados"
    r"abaixo.",
    alignment="left",
    line_spacing=0.8,
    font="CMU Serif"
).scale(0.35)

# Posiciona abaixo e alinha à esquerda com o grupo
explicacao.next_to(grupo2, DOWN, buff=0.3)
explicacao.align_to(grupo2, LEFT)  # <- ESSENCIAL!

self.play(Write(explicacao, run_time = 1))
self.wait(1)

quadrados = VGroup(*[
```

```
    VGroup(
        Square(0.2*i).next_to(DOWN, UP, buff=0)
        .shift(6*LEFT + 1.24**i*RIGHT + i*0.05*RIGHT),
        MathTex(f'{i}').scale(0.7).next_to(DOWN, DOWN, buff=0.3)\\
        .shift(6*LEFT + 1.24**i*RIGHT + i*0.05*RIGHT),
    )
    for i in range(1, 8, 2)
]).add(Tex('...').next_to(DOWN, UP, buff=0))
.shift(4.5*LEFT + 1.24**8*RIGHT + 8*0.05*RIGHT)

self.play(Write(quadrados))

self.play(quadrados.animate.shift(2*LEFT))

explicacao1 = Paragraph(
    r"Nessa sequência, a diferença constante entre os lados é 2,"
    r"que representa a razão da progressão.",
    alignment="left",
    line_spacing=0.8,
    font="CMU Serif"
).scale(0.35)

# Posiciona abaixo e alinha à esquerda com o grupo
explicacao1.next_to(grupo2, DOWN, buff=5)
explicacao1.align_to(grupo2, LEFT)  # <- ESSENCIAL!

self.play(Write(explicacao1, run_time = 1))
self.wait(1)

explicacao_formula = Paragraph(
    r"Com isso, podemos escrever uma função que calcule o tamanho \\
    do lado do n-ésimo quadrado dessa progressão.",
    alignment="left",
    line_spacing=0.8,
    font="CMU Serif"
).scale(0.35)

# Posiciona abaixo e alinha à esquerda com o grupo
explicacao_formula.next_to(grupo2, DOWN, buff=5.5)
explicacao_formula.align_to(grupo2, LEFT)  # <- ESSENCIAL!
```

```
self.play(Write(explicacao_formula, run_time = 1))
self.wait(1)

funcoes_exemplo = MathTex(r'f(1)=1\\f(2)=3\\f(3)=4\\f(4)=7').scale(0.7)
.shift(1.5*RIGHT + 0.2*DOWN)
funcao = MathTex('f(n) = 1 + 2n').scale(0.7).shift(4*RIGHT + 0.5*DOWN)

self.play(Write(funcoes_exemplo))
self.play(Write(funcao))

 # Apaga todos os elementos visíveis com fade
self.play(*[FadeOut(mob) for mob in self.mobjects])
self.wait(0.5)
```

O resultado desse código está apresentado na Figura 4.4.

Figura 4.4: Definição e exemplo de uma progressão aritmética.



Fonte: Elaborado pelo autor.

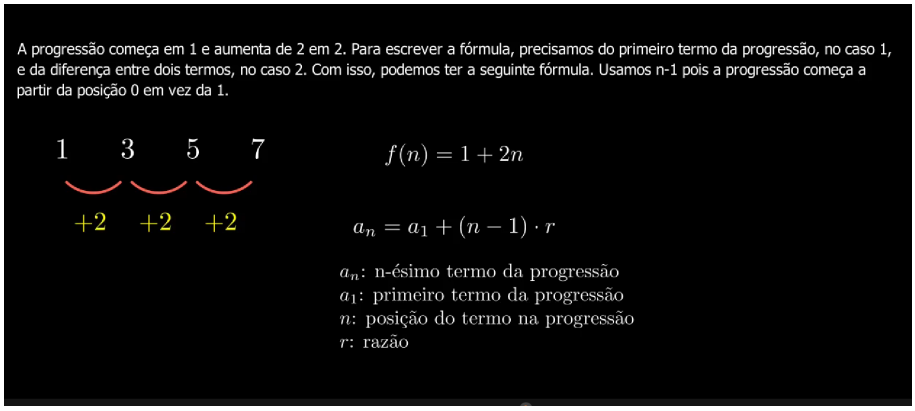
Agora, mostramos a dedução da fórmula da progressão, explorando visualmente seus termos e sua razão. Para isso, utilizamos o código a seguir. Ele utiliza a classe `Paragraph` alinhados à esquerda, controlando o espaçamento entre linhas com `line_spacing` e redimensionando via `scale`. A sequência numérica é formada com `Tex`, enquanto as setas curvas que ligam os termos são instâncias de `ArcBetweenPoints`, acompanhadas de anotações criadas com `Tex` e posicionadas por `next_to`. Para organização, todos esses elementos são agrupados em um `VGroup`. As fórmulas são apresentadas com a `MathTex` e complementadas com descrições também em TEX.

```
duvida = Paragraph(
    r"Como podemos deduzir essa função?",
```

```
    alignment="left",
    line_spacing=0.8,
    font="CMU Serif"
).scale(0.35)

self.play(Write(duvida))
self.wait(1.5)
self.play(FadeOut(duvida))

numeros = [
    Tex(f'{i}').scale(0.9).shift(2 * LEFT + i * 0.5 * RIGHT)
    for i in range(1, 8, 2)
]
steps = [
    ArcBetweenPoints(start=start.get_center() + 0.05 * RIGHT,
                     end=end.get_center() + 0.1 * LEFT,
                     color=RED).shift(0.5 * DOWN)
    for start, end in zip(numeros[:-1], numeros[1:])
]
steps_number = [
    Tex(f'+2', color=YELLOW).scale(0.8).next_to(steps[i], DOWN, buff=0.3)
    .shift(0.05 * LEFT)
    for i in range(len(steps))
]
visualizar = VGroup(*numeros, *steps, *steps_number)
visualizar.shift(UP * 1.2)

explicacao3 = Paragraph(
    r"A progressão começa em 1 e aumenta de 2 em 2. Para escrever"
    r"a fórmula, precisamos do primeiro termo da progressão, no caso 1,",
    r"e da diferença entre dois termos, no caso 2. Com isso, podemos"
    r"ter a seguinte fórmula. Usamos n-1 pois a progressão começa a ",
    r"partir da posição 0 em vez da 1.",
    alignment="left",
    line_spacing=0.8,
    font="CMU Serif"
).scale(0.35).shift(UP*2.4)

self.play(Write(explicacao3), run_time = 2)
self.wait(1.5)
```

```
self.play(Write(visualizar, run_time = 3))
self.wait(1)

self.play(visualizar.animate.shift(4.5 * LEFT), run_time = 1.5)

formula = MathTex(r'f(n) = 1 + 2n').scale(0.7).shift(UP*1.1)
self.play(Write(formula))

formula_geral = MathTex(r'a_{n} = a_{1} + (n - 1) \cdot r').scale(0.7)
self.play(Write(formula_geral))
self.wait(1)

termos = Tex(
    r'''
    \raggedright
    $a_n$: n-ésimo termo da progressão \\
    $a_1$: primeiro termo da progressão \\
    $n$: posição do termo na progressão \\
    $r$: razão
    '''
).scale(0.6).shift(1.2 * DOWN + 0.5*RIGHT)

self.play(Write(termos))
self.wait(1)

self.play(*[FadeOut(mob) for mob in self.mobjects])
self.wait(0.5)
```

Figura 4.5: Dedução da fórmula da progressão aritmética.



Fonte: Elaborado pelo autor.

4.3.2 Exercício

Neste momento, passamos a apresentar uma questão extraída do SAEPE. Primeiramente, a palavra "Questão" é adicionada à cena com a `Text` e a `Line`, cuja cor é atualizada dinamicamente por um *updater* adicionado via `add_updater`, utilizando `interpolate_color` e `np.sin`.

```
# Título com linha animada
t5 = Text("Questão", font="Fira Sans").scale(0.5)
self.play(Write(t5, run_time=2))
self.wait(0.5)

linha5 = Line(
    start=t5.get_bottom() + DOWN * 0.2 + LEFT * t5.width / 2,
    end=t5.get_bottom() + DOWN * 0.2 + RIGHT * t5.width / 2,
    stroke_width=6,
)
linha5.set_color(RED)

def atualizar_linha5(obj, dt):
    tempo = self.time
    nova_cor = interpolate_color(RED, BLUE, (np.sin(tempo * 2) + 1) / 2)
    obj.set_color(nova_cor)

linha5.add_updater(atualizar_linha5)
self.play(FadeIn(linha5))
self.wait(1)

grupo_titulo5 = VGroup(t5, linha5)
self.play(grupo_titulo5.animate.scale(0.9).to_corner(UL), run_time=1.5)
self.wait(1)
```

Enunciado

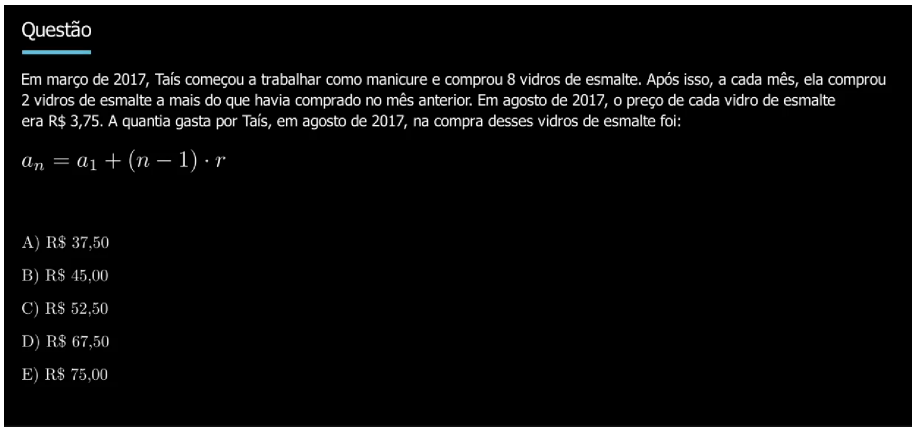
Nesta parte, o enunciado da questão é construído com a classe `Paragraph`, posicionada logo abaixo do grupo título com os métodos `next_to` e `align_to`, assim como nas outras partes anteriores. Em seguida, a fórmula geral da progressão é apresentada por meio de `MathTex`, também alinhada e posicionada em relação ao enunciado. As alternativas de resposta são criadas como um `VGroup` contendo múltiplos objetos `Tex`, organizados verticalmente com `arrange(DOWN, aligned_edge=LEFT)` para manter o alinhamento uniforme, e posicionados abaixo do enunciado usando

`.next_to` e `align_to`. As animações `Write` são aplicadas sequencialmente para a exibição gradual dos textos, com pausas controladas por `wait`.

```
# Enunciado DA QUESTÃO
enunciado = Paragraph(
    "Em março de 2017, Taís começou a trabalhar como manicure e "
    "comprou 8 vidros de esmalte. Após isso, a cada mês, ela comprou",
    "2 vidros de esmalte a mais do que havia comprado no mês anterior."
    "Em agosto de 2017, o preço de cada vidro de esmalte",
    "era R$ 3,75. A quantia gasta por Taís, em agosto de 2017, na "
    "compra desses vidros de esmalte foi:",
    alignment="left",
    line_spacing=0.8,
    font="CMU Serif"
).scale(0.35)

enunciado.next_to(grupo_titulo5, DOWN, buff=0.3)
enunciado.align_to(grupo_titulo5, LEFT)
self.play(Write(enunciado, run_time=2))
self.wait(1)

formula_geral1 = MathTex(r'a_n = a_1 + (n - 1) \cdot r').scale(0.7)
formula_geral1.next_to(enunciado, DOWN, buff=0.3)
formula_geral1.align_to(enunciado, LEFT)
self.play(Write(formula_geral1))
self.wait(1)

# Alternativas
alternativas = VGroup(
    Tex("A) R\$ 37{,}50", font_size=24),
    Tex("B) R\$ 45{,}00", font_size=24),
    Tex("C) R\$ 52{,}50", font_size=24),
    Tex("D) R\$ 67{,}50", font_size=24),
    Tex("E) R\$ 75{,}00", font_size=24),
).arrange(DOWN, aligned_edge=LEFT)

alternativas.next_to(enunciado, DOWN, buff=1.6).align_to(enunciado, LEFT)
self.play(Write(alternativas))
self.wait(1)
```

A Figura 4.6 contém o frame final da construção do enunciado.

Figura 4.6: Enunciado da questão do SAEPE sobre progressão aritmética.



Fonte: Elaborado pelo autor.

Solução

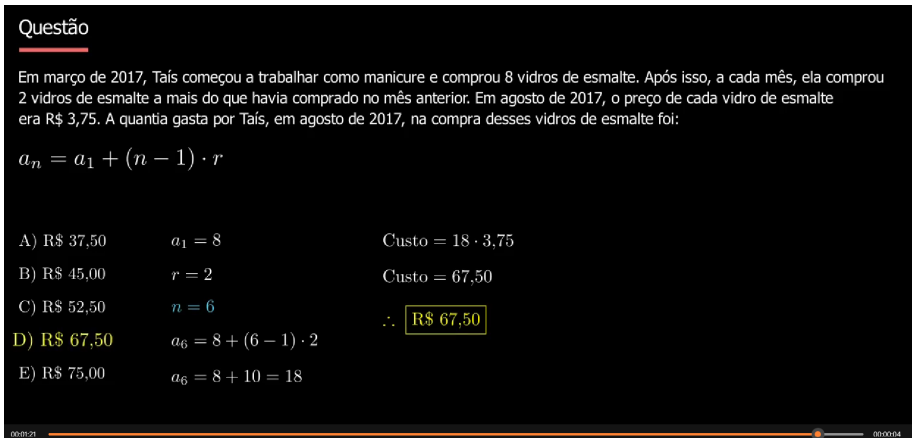
Por fim, exibimos a resolução da questão dividida em etapas, destacando os cálculos e a alternativa correta de maneira clara e didática. Cada etapa corresponde a um grupo `VGroup` contendo expressões matemáticas criadas com `MathTex`, organizadas verticalmente com `arrange`. Cada grupo é posicionado em relação às alternativas e entre si usando `next_to` e `align_to`. Para destacar a alternativa correta, são usadas as animações `Circumscribe`, que contorna o texto, `Indicate` que aumenta o tamanho e diminui em seguida, e a mudança de cor pelo método `animate.set_color`.

```
# Resolução PARTE 1 ---
parte1 = VGroup(
    MathTex(r"a_1 = 8", font_size=26),
    MathTex(r"r = 2", font_size=26),
    MathTex(r"n = 6", font_size=26, color=BLUE),
    MathTex(r"a_6 = 8 + (6 - 1) \cdot 2", font_size=26),
    MathTex(r"a_6 = 8 + 10 = 18", font_size=26),
).arrange(DOWN, aligned_edge=LEFT, buff=0.3)

parte1.next_to(alternativas, RIGHT, buff=1)
parte1.align_to(alternativas, UP)
for linha in parte1:
    self.play(Write(linha))
    self.wait(0.5)

# Resolução PARTE 2
parte2 = VGroup(
```

```
        MathTex(r"\text{Custo} = 18 \cdot 3{,}75", font_size=26),
        MathTex(r"\text{Custo} = 67{,}50", font_size=26),
        MathTex(r"\therefore\ \boxed{\text{R\$ 67{,}50}}",
        font_size=26, color=YELLOW),
    ).arrange(DOWN, aligned_edge=LEFT, buff=0.3)

    parte2.next_to(parte1, RIGHT, buff=1)
    parte2.align_to(parte1, UP)
    for linha in parte2:
        self.play(Write(linha))
        self.wait(0.5)

    #Destacar alternativa correta (letra D)
    alternativa_correta = alternativas[3]
    self.play(Circumscribe(alternativa_correta, color=YELLOW))
    self.play(Indicate(alternativa_correta, color=YELLOW))
    self.play(alternativa_correta.animate.set_color(YELLOW))
    self.wait(3)
```

A Figura 4.7 mostra o momento final da solução.

Figura 4.7: Momento final da apresentação da solução de um exercício sobre progressão aritmética.



Fonte: Elaborado pelo autor.

4.4 Área do trapézio

Dando continuidade, além da progressão aritmética abordada anteriormente, apresentamos agora o código do tema da Área do Trapézio que se relaciona ao campo da Geometria. Trata-se da habilidade D12, que orienta o estudante a resolver problemas envolvendo área de figuras planas.

No caso específico do trapézio, espera-se que o aluno compreenda sua estrutura geométrica e utilize adequadamente a fórmula da área, relacionando as medidas das bases e da altura à situação descrita no enunciado. Essa habilidade envolve não apenas o uso correto da fórmula, mas também a interpretação dos dados apresentados e o raciocínio necessário para aplicá-los em um contexto prático.

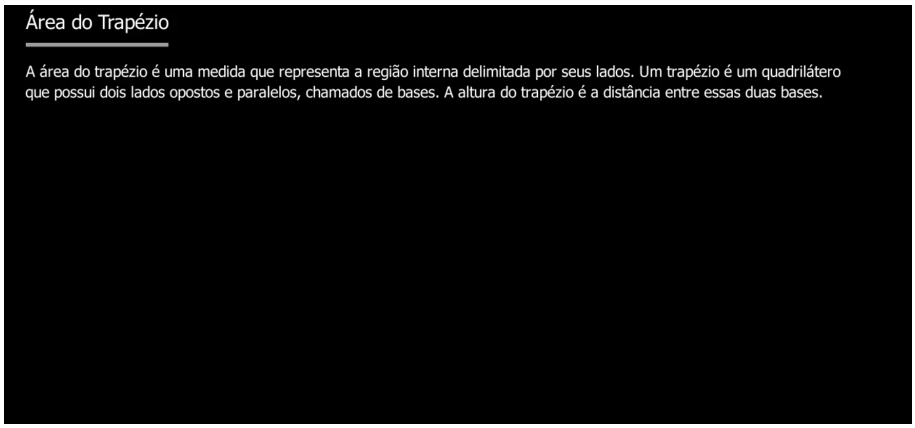
4.4.1 Fundamentação

Abaixo consta o início do vídeo com o tema proposto.

```
# Texto explicativo inicial
explicacao = Paragraph(
    r"   A área do trapézio é uma medida que representa a região interna",
    r"delimitada por seus lados. Um trapézio é um quadrilátero que possui",
    r"dois lados opostos e paralelos, chamados de bases. A altura do ",
    r"trapézio é a distância entre essas duas bases.",
    r"",
    alignment="left",
    line_spacing=0.8, font="CMU Serif"
).scale(0.35)

# Posiciona o parágrafo abaixo de algum grupo anterior
explicacao.next_to(grupo2, DOWN, buff=0.3)
explicacao.align_to(grupo2, LEFT)

self.play(Write(explicacao, run_time=1))
self.wait(1)
```

Um título menor é criado com `Text()` e colocado no topo esquerdo com `.to_corner()`. A linha é no mesmo formatos das feitas anteriores, que muda a cor com o passar do tempo. A linha e o título são agrupados em `VGroup` e animados juntos. Um texto explicativo é criado com `Paragraph()` com alinhamento à esquerda, espaçamento entre linhas e posicionado próximo ao título com `.next_to()` e `.align_to()`.

Figura 4.8: Texto explicativo inicial sobre a área do trapézio.

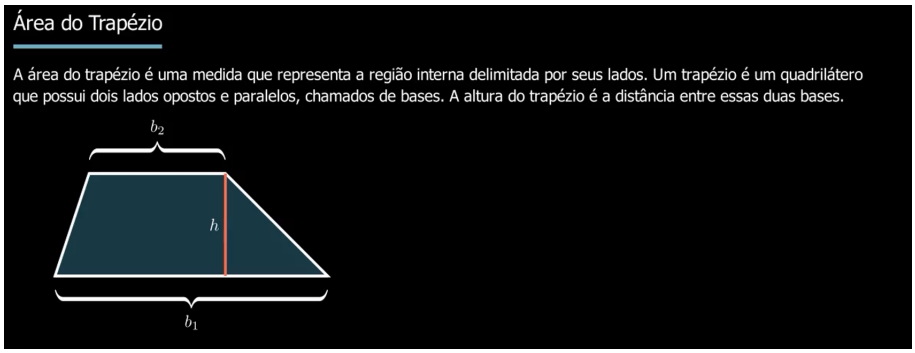


Fonte: Elaborado pelo autor.

```
# DEFINIÇÃO DOS PONTOS DO TRAPÉZIO ORIGINAL
pontos10 = [[-6, -0.4, 0], [-2, -0.4, 0], [-3.5, 1.1, 0],
[-5.5, 1.1, 0]]
trapezio10 = Polygon(*pontos10, color=WHITE, fill_color=BLUE,
fill_opacity=0.3)
base_maior10 = Line(pontos10[0], pontos10[1], color=WHITE)
base_menor10 = Line(pontos10[3], pontos10[2], color=WHITE)
altura10 = Line(pontos10[2], [-3.5, -0.4, 0], color=RED)
brace_b110 = Brace(base_maior10, DOWN)
label_b110 = MathTex("b_1", font_size=24).next_to(brace_b110,
DOWN, buff=0.1)
brace_b210 = Brace(base_menor10, UP)
label_b210 = MathTex("b_2", font_size=24).next_to(brace_b210,
UP, buff=0.1)
label_h10 = MathTex("h", font_size=24).next_to(altura10, LEFT,
buff=0.1)

#CONSTRUÇÃO DO TRAPÉZIO
self.play(Create(trapezio10))
self.play(Create(base_maior10), Create(base_menor10))
self.play(Create(altura10), Write(label_h10))
self.play(GrowFromCenter(brace_b110), Write(label_b110))
self.play(GrowFromCenter(brace_b210), Write(label_b210))
```

O trapézio é definido com pontos numéricos, desenhado com `Polygon()`, e suas bases e altura são criadas com `Line()` e rotuladas com `Brace()` e `MathTex()`. Tudo é animado sequencialmente com `Create()` e `Write()`.

Figura 4.9: Criação do trapézio



Fonte: Elaborado pelo autor.

```
explicacao1 = Paragraph(
    r"   Para calcular a área de um trapézio, utilizamos a média",
    r"aritmética das bases multiplicada pela altura.A fórmula é ",
    r"dada pela expressão abaixo:",
    r"",
    alignment="left",
    line_spacing=0.8, font="CMU Serif"
).scale(0.35)

# Posiciona o parágrafo abaixo de algum grupo anterior
explicacao1.next_to(label_b110, DOWN, buff=0.3)
explicacao1.align_to(grupo2, LEFT)
self.play(Create(explicacao1))
self.wait(1)

#FÓRMULA INICIAL COM TRAPÉZIO
trap_copia10 = trapezio10.copy().scale(0.25).set_fill(BLUE,
opacity=0.5)
inicio10 = MathTex(r"\text{Área}(", font_size=30)
igual10 = MathTex(r")\ =", font_size=30)
formula_inicial10 = VGroup(inicio10, trap_copia10, igual10)
.arrange(RIGHT, buff=0.2)
formula_inicial10.to_corner(DOWN + LEFT, buff=0.8)

trap_anim10 = trapezio10.copy()
trap_anim10.generate_target()
trap_anim10.target.scale(0.25)
trap_anim10.target.move_to(trap_copia10.get_center())

self.play(Write(inicio10))
```

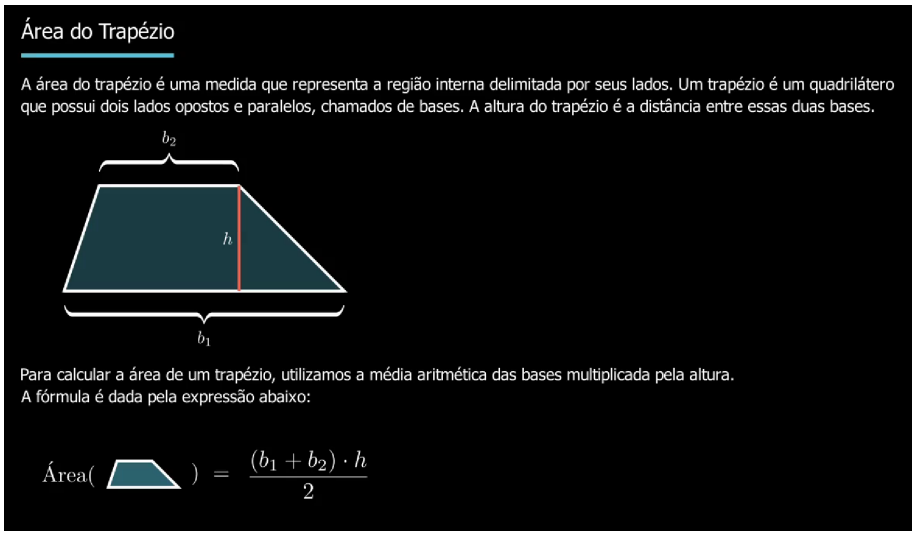
```
self.play(MoveToTarget(trap_anim10), run_time=2)
self.play(Write(igual10))
self.remove(trap_anim10)
self.add(trap_copia10)
self.wait()

#Fórmula final em LaTeX ao lado do símbolo de "="
formula_area10 = MathTex(
    r"\frac{(b_1 + b_2) \cdot h}{2}", font_size=32
)

# Posiciona ao lado do sinal de igual
formula_area10.next_to(igual10, RIGHT, buff=0.3)

# Adiciona na cena com animação
self.play(Write(formula_area10))
self.wait(2)

# Agrupa a expressão completa se quiser reposicionar depois
expressao_completa = VGroup(inicio10, trap_copia10, igual10,
formula_area10)

#Apaga todos os elementos visíveis com fade
self.play(*[FadeOut(mob) for mob in self.mobjects])
self.wait(0.5)
```

De início é mostrado um texto usando o `Paragraph()`, em seguida a fórmula da área do trapézio é apresentada utilizando a classe `MathTex` para renderizar a expressão matemática:

$$\frac{(b_1 + b_2) \cdot h}{2}$$

É criada também uma cópia reduzida do trapézio original usando os métodos `copy()` e `scale()`, posicionada ao lado da fórmula. Usamos o método `MoveToTarget()` para transformar suavemente o trapézio maior em sua versão reduzida. Por fim, a fórmula é exibida gradualmente com a animação `Write()`.

Figura 4.10: Fórmula da área do trapézio animada



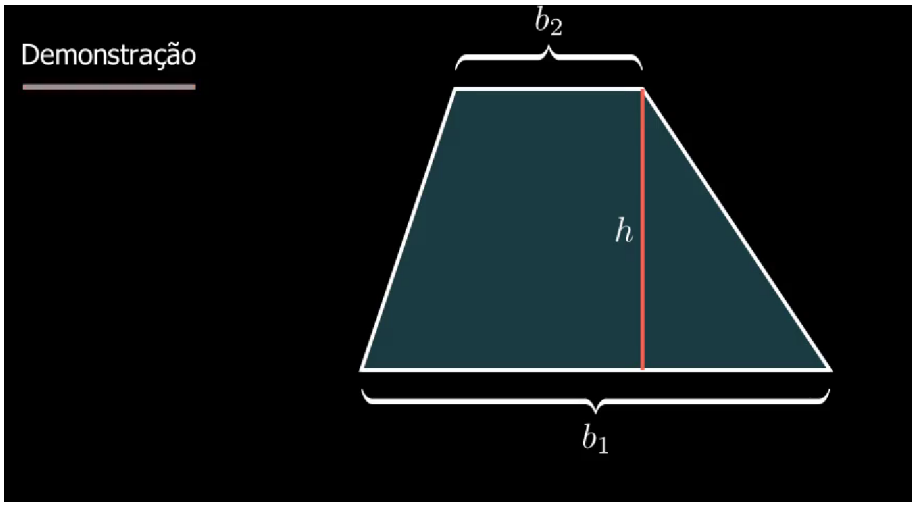
Fonte: Elaborado pelo autor.

```
# Título com linha animada - DEMONSTRAÇÃO (Variáveis com índice 6)
t6 = Text("Demonstração", font="Fira Sans").scale(0.5)
self.play(Write(t6, run_time=2))
self.wait(0.5)

linha6 = Line(
    start=t6.get_bottom() + DOWN * 0.2 + LEFT * t6.width / 2,
    end=t6.get_bottom() + DOWN * 0.2 + RIGHT * t6.width / 2,
    stroke_width=6,
)
linha6.set_color(RED)

def atualizar_linha6(obj, dt):
    tempo = self.time
    nova_cor = interpolate_color(RED, BLUE,
    (np.sin(tempo * 2) + 1) / 2)
    obj.set_color(nova_cor)

linha6.add_updater(atualizar_linha6)
self.play(FadeIn(linha6))
self.wait(1)

grupo_titulo6 = VGroup(t6, linha6)
self.play(grupo_titulo6.animate.scale(0.9).to_corner(UL),
run_time=1.5)
self.wait(1)
```

```
#DEFINIÇÃO DOS PONTOS DO TRAPÉZIO ORIGINAL
pontos = [[-3, 0, 0], [2, 0, 0], [0, 3, 0], [-2, 3, 0]]
trapezio = Polygon(*pontos, color=WHITE, fill_color=BLUE,
fill_opacity=0.3)
base_maior = Line(pontos[0], pontos[1], color=WHITE)
base_menor = Line(pontos[3], pontos[2], color=WHITE)
altura = Line(pontos[2], [0, 0, 0], color=RED)
brace_b1 = Brace(base_maior, DOWN)
label_b1 = MathTex("b_1", font_size=36).next_to(brace_b1,
DOWN, buff=0.1)
brace_b2 = Brace(base_menor, UP)
label_b2 = MathTex("b_2", font_size=36).next_to(brace_b2,
UP, buff=0.1)
label_h = MathTex("h", font_size=36).next_to(altura, LEFT,
buff=0.1)

#CONSTRUÇÃO DO TRAPÉZIO
self.play(Create(trapezio))
self.play(Create(base_maior), Create(base_menor))
self.play(Create(altura), Write(label_h))
self.play(GrowFromCenter(brace_b1), Write(label_b1))
self.play(GrowFromCenter(brace_b2), Write(label_b2))
```

Nesta parte do código, aplicamos os conceitos e recursos apresentados anteriormente como `Text, Write, Line`, entre outros. O título e a linha são agrupados em `VGroup` e reposicionados com `.to_corner()`. Em seguida, o trapézio é feito usando a classe `Polygon()`, e suas bases e altura são representadas com `Line()`, com rótulos criados com `Brace()` e `MathTex()`.

Figura 4.11: Trapézio para demonstração



Fonte: Elaborado pelo autor.

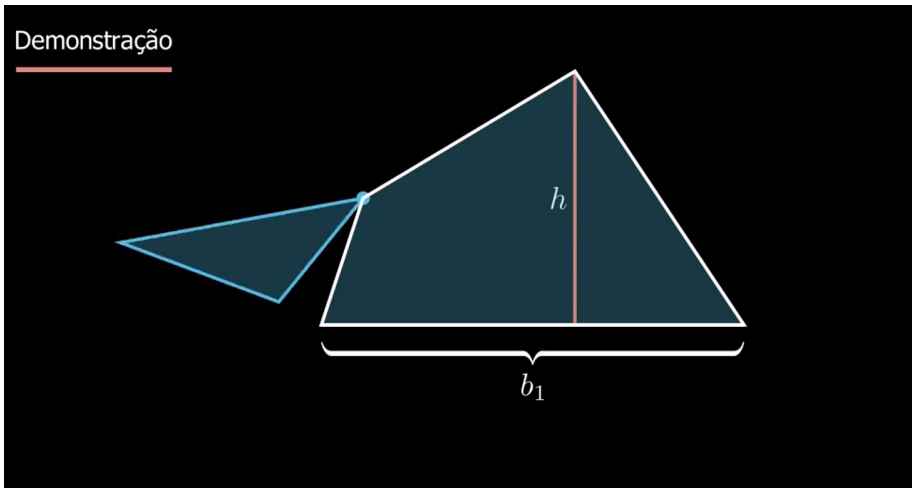
```
# FÓRMULA INICIAL COM TRAPÉZIO
trap_copia = trapezio.copy().scale(0.25).set_fill(BLUE,
opacity=0.5)
inicio = MathTex(r"\text{Área}(", font_size=36)
igual = MathTex(r")\ =", font_size=36)
formula_inicial = VGroup(inicio, trap_copia, igual)
.arrange(RIGHT, buff=0.2)
formula_inicial.to_corner(DOWN + LEFT, buff=0.8)

trap_anim = trapezio.copy()
trap_anim.generate_target()
trap_anim.target.scale(0.25)
trap_anim.target.move_to(trap_copia.get_center())

self.play(Write(inicio))
self.play(MoveToTarget(trap_anim), run_time=2)
self.play(Write(igual))
self.remove(trap_anim)
self.add(trap_copia)
self.wait()

# === 4. ROTAÇÃO DO SEGMENTO AE ===
A_coords = [-3, 0, 0]
E_coords = [-2.505, 1.5, 0]
D_coords = [-2, 3, 0]
C_coords = [0, 3, 0]
```

```
ponto_A = Dot(A_coords, color=RED)
ponto_E = Dot(E_coords, color=BLUE)
segmento_AE = Line(A_coords, E_coords, color=RED)
self.play(Create(ponto_E))
self.play(Create(segmento_AE), FadeIn(ponto_A))
grupo = VGroup(segmento_AE, ponto_A)

trapezio1 = Polygon(A_coords, [2, 0, 0], C_coords, E_coords,
color=WHITE, fill_color=BLUE, fill_opacity=0.3)
self.play(Rotate(grupo, angle=-PI, about_point=ponto_E.get_center()))
self.wait()

triangulo_EDC = Polygon(E_coords, D_coords, C_coords, color=BLUE,
fill_color=BLUE, fill_opacity=0.3)
self.play(Create(triangulo_EDC), FadeOut(VGroup(segmento_AE,
base_menor, brace_b2, label_b2)),
          FadeIn(trapezio1), FadeOut(trapezio), FadeOut(ponto_A))
self.wait()
self.play(Rotate(triangulo_EDC, angle=PI,
about_point=ponto_E.get_center()))
self.wait()

#BRACE NOVO PARA BASE b_2
ponto_inicio = [-5, 0, 0]
ponto_fim = [-3, 0, 0]
base_manual = Line(ponto_inicio, ponto_fim, color=WHITE)
brace_b2_manual = Brace(base_manual, direction=DOWN)
label_b2_manual = MathTex("b_2", font_size=36).next_to(brace_b2_manual,
DOWN, buff=0.2)
self.play(Create(base_manual))
self.play(GrowFromCenter(brace_b2_manual), Write(label_b2_manual))
self.wait()

grupo10 = VGroup(brace_b2_manual, label_b2_manual)
ponto_inicio1 = [-5, 0, 0]
ponto_fim1 = [2, 0, 0]
base_manual1 = Line(ponto_inicio1, ponto_fim1, color=WHITE)
brace_b2_manual1 = Brace(base_manual1, direction=DOWN)
label_b2_manual1 = MathTex("(b_1 + b_2)", font_size=36)
.next_to(brace_b2_manual1, DOWN, buff=0.2)
grupo11 = VGroup(brace_b2_manual1, label_b2_manual1)
```

```
self.play(FadeOut(brace_b1), FadeOut(label_b1), FadeOut(ponto_E))

self.play(ReplacementTransform(grupo10, grupo11))
```

No trecho do código acima, cria-se uma cópia menor do trapézio original com `copy()` e `scale()`, e para deixar com coloração azul, mas transparente, usamos `set_fill()`. Depois, o trapézio original é transformado em sua versão menor usando `generate_target()` e `MoveToTarget()`. O trapézio grande é removido e substituído pela cópia criada. Os pontos e o segmento AE são criados e animados com (`Create()`, `FadeIn()`). O grupo formado por AE e o ponto A é rotacionado em torno do ponto E com `Rotate()`. Um triângulo substitui parte do trapézio, enquanto elementos antigos, como a base menor e seus braces, são removidos com `FadeOut()`. O triângulo também é rotacionado . Por fim, novas bases são feitas manualmente com `Line()` e destacadas por `Brace()` e `MathTex()` para indicar b_2 e $b_1 + b_2$. Para juntar os dois braces mostrando a soma das bases, usamos o `ReplacementTransform()`.

Figura 4.12: Parte da construção usada na demonstração da fórmula do trapézio.



Fonte: Elaborado pelo autor.

```
#TRIÂNGULO FINAL E FÓRMULA COMPLETA
tri_real = Polygon([-5, 0, 0], [2, 0, 0], [0, 3, 0], color=YELLOW,
fill_color=YELLOW, fill_opacity=0.1)
self.play(Create(tri_real))

tri_copia = tri_real.copy().scale(0.25)
area_triangulo = MathTex(r"\text{Área}(", font_size=36)
fecha = MathTex(") = ", font_size=36)
formula_final = VGroup(area_triangulo, tri_copia, fecha)
```

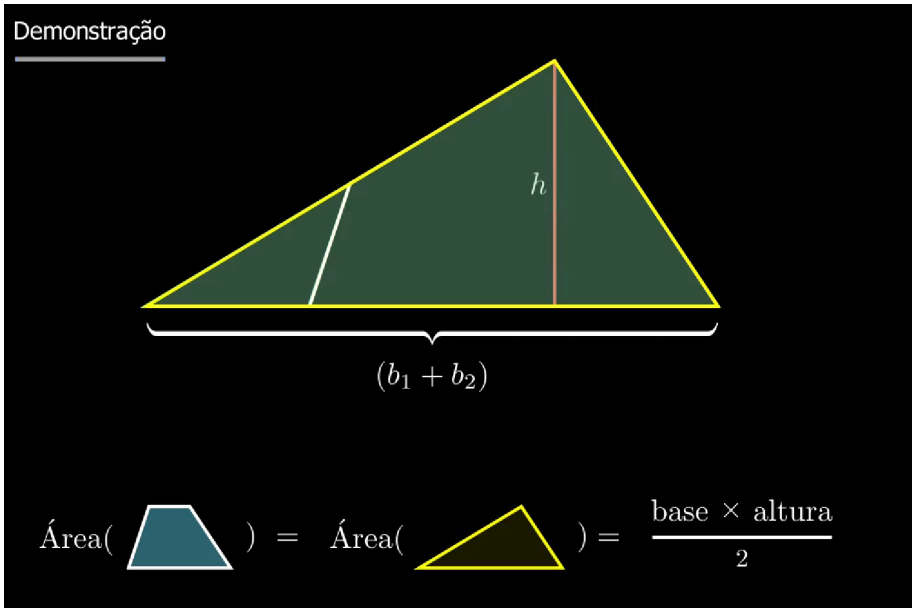
```
.arrange(RIGHT, buff=0.2)
formula_final.next_to(formula_inicial, RIGHT, buff=0.4)

tri_anim = tri_real.copy()
tri_anim.generate_target()
tri_anim.target.scale(0.25)
tri_anim.target.move_to(tri_copia.get_center())

self.play(MoveToTarget(tri_anim), run_time=2)
self.play(Write(area_triangulo), Write(fecha))
self.remove(tri_anim)
self.add(tri_copia)

#ADICIONANDO EXPRESSÃO COM "base × altura"
base1 = MathTex(r"\text{base}", font_size=36)
produto = MathTex(r"\times", font_size=34)
altura1 = MathTex(r"\text{altura}", font_size=36)

formula_final1 = VGroup(base1, produto, altura1)
.arrange(RIGHT, buff=0.2)
formula_final1.next_to(formula_final, RIGHT, buff=0.4)
formula_final1.shift(UP * 0.33)

# Linha de divisão e denominador
largura_linha = 2.2
linha_divisao = Line(LEFT * largura_linha / 2, RIGHT * largura_linha / 2)
linha_divisao.next_to(formula_final1, DOWN, buff=0.2)
denominador = MathTex("2", font_size=30)
denominador.next_to(linha_divisao, DOWN, buff=0.15)

grupo_expressao = VGroup(formula_final1, linha_divisao,
denominador)

self.play(Create(formula_final1), Create(linha_divisao),
Write(denominador))
self.wait(1)

# Cópias dos rótulos originais
b1b2_copia = label_b2_manual1.copy()
h_copia = label_h.copy()
self.add(b1b2_copia, h_copia)
```

```
#Definir distâncias personalizadas
deslocamento_b1b2_x = 0.4
distancia_desejada_produto = 0.9
distancia_desejada_h = 0.35

#Calcular posições finais
destino_b1b2 = base1.get_center() + RIGHT * deslocamento_b1b2_x
destino_produto = destino_b1b2 + RIGHT * distancia_desejada_produto
destino_h = destino_produto + RIGHT * distancia_desejada_h

#Posição inicial do produto: mais próximo de b1b2
produto.move_to(destino_b1b2 + RIGHT * 0.2)  # ponto de partida do ×
self.add(produto)

#Animação sincronizada com movimento real do ×
self.play(
    FadeOut(base1),
    FadeOut(altura1),
    b1b2_copia.animate.move_to(destino_b1b2),
    produto.animate.move_to(destino_produto),
    h_copia.animate.move_to(destino_h)
)
self.wait(0.5)

#Agrupar nova expressão com linha e denominador
nova_expressao = VGroup(b1b2_copia, produto, h_copia)
nova_expressao_with_div = VGroup(nova_expressao, linha_divisao,
denominador)
self.wait(2)

#Apaga todos os elementos visíveis com fade
self.play(*[FadeOut(mob) for mob in self.mobjects])
self.wait(0.5)
```

Nesta parte do código, as classes e métodos usados já foram mostrados anteriormente, devido a isso, omitiremos boa parte. O triângulo final e sua cópia são criados igualmente os feitos anteriormente. Para animar a transição do triângulo maior para a versão reduzida, é usada a função `generate_target()`, e `MoveToTarget()` move o triângulo até a posição desejada. Cópias dos rótulos originais das bases e altura são feitas com `copy()` e adicionadas à cena.As posições finais desses rótulos e do símbolo de multiplicação são calculadas usando `get_center()`, e somadas a vetores

para posicioná-los corretamente na cena. Por fim, todos os elementos visíveis são removidos com `FadeOut()` para preparar a cena para o próximo conteúdo.

Figura 4.13: Etapa final da demonstração sobre a fórmula da área do trapézio.



Fonte: Elaborado pelo autor.

4.4.2 Exercício

Enunciado

O título "Questão" é criado da mesma forma que os já feitos nos códigos anteriores. Em seguida, o enunciado é criado com `Paragraph`. O trapézio é desenhado com `Polygon`, suas bases e altura são representadas por `Line` e recebem rótulos com `Tex`. Todos esses elementos são agrupados com `VGroup` e animados para aparecer. Por fim, as alternativas são criadas com `Tex`, organizadas verticalmente com `VGroup.arrange()`, posicionadas abaixo do trapézio.

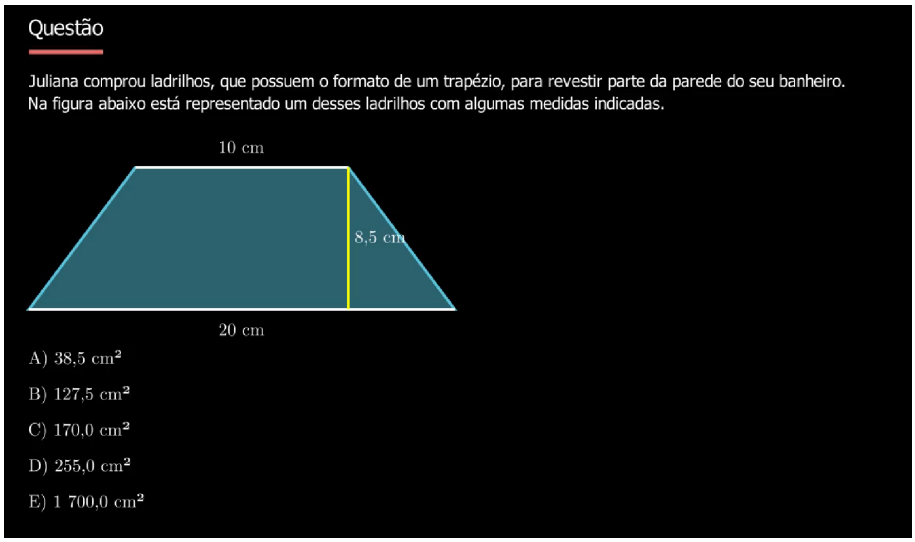
```
# Título com linha animada
t5 = Text("Questão", font="Fira Sans").scale(0.5)
self.play(Write(t5, run_time=2))
self.wait(0.5)

linha5 = Line(
    start=t5.get_bottom() + DOWN * 0.2 + LEFT * t5.width / 2,
    end=t5.get_bottom() + DOWN * 0.2 + RIGHT * t5.width / 2,
    stroke_width=6,
)
linha5.set_color(RED)
```

```
def atualizar_linha5(obj, dt):
    tempo = self.time
    nova_cor = interpolate_color(RED, BLUE, (np.sin(tempo * 2) + 1) / 2)
    obj.set_color(nova_cor)

linha5.add_updater(atualizar_linha5)
self.play(FadeIn(linha5))
self.wait(1)

grupo_titulo5 = VGroup(t5, linha5)
self.play(grupo_titulo5.animate.scale(0.9).to_corner(UL), run_time=1.5)
self.wait(1)

# Enunciado
enunciado = Paragraph(
    "Juliana comprou ladrilhos, que possuem o formato de um trapézio,",
    "para revestir parte da parede do seu banheiro. Na figura abaixo ",
    "está representado um desses ladrilhos com algumas medidas indicadas.",
    alignment="left",
    line_spacing=0.8,
    font="CMU Serif"
).scale(0.35)

enunciado.next_to(grupo_titulo5, DOWN, buff=0.3)
enunciado.align_to(grupo_titulo5, LEFT)
self.play(Write(enunciado, run_time=2))
self.wait(1)

#DESENHO DO TRAPÉZIO E RÓTULOS
A = [-3, 0, 0]
B = [3, 0, 0]
C = [1.5, 2, 0]
D = [-1.5, 2, 0]
trapezio = Polygon(A, B, C, D, color=BLUE, fill_opacity = 0.5)

base_maior = Line(A, B)
rotulo_b2 = Tex("20 cm", font_size=24).next_to(base_maior, DOWN, buff=0.2)

base_menor = Line(D, C)
rotulo_b1 = Tex("10 cm", font_size=24).next_to(base_menor, UP, buff=0.2)

ponto_E = [C[0], A[1], 0]
```

```
altura = Line(C, ponto_E, color=YELLOW)
rotulo_h = Tex("8{,}5 cm", font_size=24).next_to(altura, RIGHT, buff=0.1)

# Agrupar tudo para mover junto
trapezio_group = VGroup(trapezio, base_maior, base_menor, altura,
rotulo_b1, rotulo_b2, rotulo_h)
trapezio_group.next_to(enunciado, DOWN, buff=0.4)
trapezio_group.align_to(enunciado, LEFT)

self.play(Create(trapezio), Create(base_maior), Write(rotulo_b2),
          Create(base_menor), Write(rotulo_b1),
          Create(altura), Write(rotulo_h))
self.wait(1)

#ALTERNATIVAS
alternativas = VGroup(
    Tex("A) 38,5 cm²", font_size=24),
    Tex("B) 127,5 cm²", font_size=24),
    Tex("C) 170,0 cm²", font_size=24),
    Tex("D) 255,0 cm²", font_size=24),
    Tex("E) 1 700,0 cm²", font_size=24),
).arrange(DOWN, aligned_edge=LEFT)

alternativas.next_to(trapezio_group, DOWN, buff=0.2).align_to(enunciado, LEFT)
self.play(Write(alternativas))
self.wait(1)
```

O resultado da construção do enunciado está apresentado na Figura 4.14.

Figura 4.14: Enunciado da Questão X do SAEPE X evolvendo área do trapézio.



Fonte: Elaborado pelo autor.

Solução

A resolução é mostrada com expressões matemáticas (`MathTex`) organizadas verticalmente e posicionadas ao lado das alternativas. Cada parte é escrito com animação `Write()`. Por fim, a alternativa correta é destacada visualmente com as animações `Circumscribe()` e `Indicate()`, além de mudar a cor do texto para amarelo, reforçando a resposta certa.

```
# RESOLUÇÃO DA QUESTÃO
linha1 = MathTex(r"A = \frac{(b_1 + b_2) \cdot h}{2}", font_size=26,
color = RED)
linha2 = MathTex(r"A = \frac{(10 + 20) \cdot 8{,}5}{2}", font_size=26,
color = RED)
linha3 = MathTex(r"A = \frac{30 \cdot 8{,}5}{2}", font_size=26,
color = RED)
linha4 = MathTex(r"A = \frac{255}{2}", font_size=26, color = RED)
linha5 = MathTex(r"A = 127{,}5", font_size=26, color = RED)
linha6 = MathTex(r"\therefore\ \boxed{A = 127{,}5\ \text{cm}^2}",
font_size=26, color = RED)

resolucao = VGroup(linha1, linha2, linha3, linha4, linha5, linha6)
.arrange(DOWN, aligned_edge=LEFT, buff=0.2)
resolucao.move_to(alternativas[0].get_center()).align_to(alternativas,
LEFT).shift(RIGHT * 7.2)

for linha in resolucao:
```

```
        self.play(Write(linha))
        self.wait(0.5)

    #DESTACAR ALTERNATIVA CORRETA (letra B)
    alternativa_correta = alternativas[1]
    self.play(Circumscribe(alternativa_correta, color=YELLOW))
    self.play(Indicate(alternativa_correta, color=YELLOW))
    self.play(alternativa_correta.animate.set_color(YELLOW))
    self.wait(3)
```

Capítulo 5

Considerações finais

Uma das grandes dificuldades no ensino da matemática, em especial de assuntos ligados à geometria, está na predominância de metodologias tradicionais, como a memorização e abstração sem visualização. Essa abordagem, muitas vezes, não favorece o desenvolvimento de habilidades como raciocínio lógico e percepção geométrica, que são essenciais para resolução de questões e entendimento do assunto.

Com essa realidade em mente, este trabalho propôs o uso da biblioteca Manim como recurso tecnológico para potencializar o ensino da matemática, explorando suas possibilidades para criar animações que ilustram conceitos, demonstram propriedades e simulam a resolução de problemas de avaliações externas, como o SAEPE.

No cenário educacional que estamos inseridos atualmente, marcado por avanços tecnológicos significativos, a matemática demanda um novo olhar de sua prática pedagógica, no qual teoria, prática e aplicação se articulem para promover experiências de aprendizagem mais dinâmicas e conceituais. Sob essa perspectiva, a escolha e uso da biblioteca Manim configura-se como uma estratégia inovadora e promissora, capaz de produzir representações visuais de grande qualidade, fomentando o raciocínio lógico, estimulando o pensamento computacional e ampliando a capacidade de abstração dos conteúdos matemáticos.

Embora o domínio da biblioteca Manim seja um desafio inicial para professores e estudantes que têm seu primeiro contato com a ferramenta, esse processo de aprendizagem torna-se também uma oportunidade formativa. Diante disso, o potencial didático do Manim manifesta-se de forma expressiva, favorecendo a contextualização dos conteúdos, a interdisciplinaridade e a autonomia na produção de materiais educativos.

É nossa expectativa que este trabalho de pesquisa promova o uso de novas práticas educacionais pelos docentes do ensino de matemática. Almejamos que os estudantes, ao interagir com esses recursos, percebam a disciplina não apenas como um conjunto de fórmulas e teorias abstratas, mas como um algo vivo, dinâmico e

profundamente conectado à realidade e tecnologia, capaz de despertar o interesse pela matemática.

Referências

AINSWORTH, S. How do animations influence learning? In: RECENT Innovations in Educational Technology that Facilitate Student Learning. [S.l.]: Emerald Publishing Limited, 2008. cap. 3, p. 37–67. (Current Perspectives on Cognition, Learning and Instruction).

ALVES, Marcia M; BATTAIOLA, André L; SPINILLO, Carla. Design de animações educacionais: levantamento de situações e motivos. **Estudos em Design**, v. 22, n. 1, 2014. DOI: 10.35522/eed.v22i1.149.

BORBA, Marcelo de Carvalho; PENTEADO, Miriam Godoy. **Informática e educação matemática**. 5. ed. Belo Horizonte: Autêntica, 2016. (Coleção Tendências em Educação Matemática).

CASTILLO, Luis Andrés; SÁNCHEZ, Ivonne C. Projeto AM^2: Animações Matemáticas com Manim. **Paradigma**, v. 46, n. 1, p. 1–16, 2025. DOI: 10.37618/PARADIGMA.1011-2251.2025.e2025009.id1623.

FUNIBER. **A importância do vídeo como ferramenta de aprendizagem**. 2019. Disponível em: <https://blogs.funiber.org/pt/formacao-professores/2019/01/12/funiber-importancia-video-ferramenta-aprendizagem>. Acesso em: 21 mar. 2025.

G1. **Ranking da educação: Brasil está nas últimas posições no Pisa 2022; veja notas de 81 países em matemática, ciências e leitura**. 2023. Disponível em: <https://g1.globo.com/educacao/noticia/2023/12/05/ranking-da-educacao-brasil-esta-nas-ultimas-posicoes-no-pisa-2022-veja-notas-de-81-paises-em-matematica-ciencias-e-leitura.ghtml>. Acesso em: 20 jun. 2025.

GEOGEBRA. **GeoGebra – Software de Matemática Dinâmica**. 2025. Disponível em: <https://www.geogebra.org/>. Acesso em: 1 abr. 2025.

KHAN ACADEMY. **Khan Academy**. Disponível em: <https://www.khanacademy.org/>. Acesso em: 20 mar. 2025.

KISHIMOTO, Eric Satoshi Suzuki; COLUCI, Vitor Rafael. Animações para o ensino de matemática usando o Manim–Python. **Professor de Matemática Online**, v. 11, n. 1, 2023. DOI: 10.21711/2319023x2023/pmo1104.

LIMA, Marta Gomes; ROCHA, Adriano Aparecido Soares da. As tecnologias digitais no ensino de matemática. **Revista Ibero-Americana de Humanidades, Ciências e Educação**, v. 8, n. 5, p. 729–739, 2022. DOI: 10.51891/rease.v8i5.5513.

MAYER, Richard E; MORENO, Roxana. Animation as an aid to multimedia learning. **Educational Psychology Review**, v. 14, n. 1, p. 87–99, 2002. DOI: 10.1023/A:1013184611077.

MORAN, J. M. **Educação inovadora: os desafios para um novo tempo**. São Paulo: Papirus, 2013.

NÓVOA, António. Formação de professores e profissão docente. In: NÓVOA, A. (Ed.). **Os professores e a sua formação**. Lisboa: Dom Quixote, 1992. v. 1. (Temas de Educação). P. 13–33.

PEARSON. **Meeting the Expectations of Gen Z in Higher Education**. 2018. Disponível em: <https://www.pearson.com/content/dam/one-dot-com/one-dot-com/us/en/files/PSONA5646-8150_TIDL_GenZ_Infographic_Print_FINAL.pdf>. Acesso em: 22 mar. 2025.

PHET. **PhET Interactive Simulations**. Disponível em: <https://phet.colorado.edu/>. Acesso em: 22 mar. 2025.

ROCATO, Paulo Sérgio. **As concepções dos professores sobre o uso de vídeos como potencializadores do processo de ensino e aprendizagem**. 2009. Dissertação de Mestrado – Universidade Cruzeiro do Sul.

SILVA, Ana Maria da. **O vídeo como recurso didático no ensino de matemática**. 2011. Dissertação de Mestrado – Universidade Federal de Goiás.

SILVA, Michelsch João da; FELCHER, Carla Denize Ott; FOLMER, Vanderlei. A produção de vídeos de matemática pelos estudantes: uma prática alinhada á educação 5.0. **Revista Eletrônica de Educação Matemática**, v. 19, p. 1–21, 2024. DOI: 10.5007/1981-1322.2024.e99268.

TAYLOR, M; POUNTNEY, D; MALABAR, I. Animation as an aid for the teaching of mathematical concepts. **Journal of Further and Higher Education**, v. 31, n. 3, p. 249–261, 2007. DOI: 10.1080/03098770701424975.

VALENTE, José Armando; FREIRE, Fernanda Maria Pereira; ARANTES, Flávia Linhalis (Ed.). **Tecnologia e Educação: passado, presente e o que está por vir**. Campinas: NIED/UNICAMP, 2018.

VELASCO, Alejandra Meraz. **O uso dos resultados das avaliações de aprendizagem no planejamento de políticas educacionais no Brasil**. Buenos Aires, 2022. Disponível em: <https://unesdoc.unesco.org/ark:/48223/pf0000379597_por>. Acesso em: 1 abr. 2025.