



**julia**  
com física  
uma introdução

Adeil Araújo e Meirivâni Oliveira



**CEARÁ**  
GOVERNO DO ESTADO  
SECRETARIA DA EDUCAÇÃO





**CEARÁ**  
GOVERNO DO ESTADO  
SECRETARIA DA EDUCAÇÃO

# Julia com Física

Uma Introdução

Versão 1.5.1

Adeil Araújo e Meirivâni Oliveira

1ª Edição



Fortaleza - Ceará  
2023

Arte da Capa e contracapa  
Meirivani Meneses de Oliveira

**Projeto Gráfico**



Dados Internacionais de Catalogação na Publicação (CIP)

D658j Araújo, Adeil  
Julia com Física - uma introdução [recurso eletrônico] / Adeil  
Araújo; Meirivâni Oliveira. - Fortaleza: SEDUC, 2023.  
  
114 p.  
  
**ISBN 978-85-8171-275-8 (E-book)**  
  
1. Linguagem - sintaxe. 2. Linguagem - programação. I. Araújo,  
Adei. II. Oliveira, Meirivâni. III. Título.  
  
CDD: 005



**Elmano de Freitas da Costa**  
Governador

**Jade Afonso Romero**  
Vice-Governadora

**Eliana Nunes Estrela**  
Secretária da Educação

**Emanuelle Grace Kelly Santos de Oliveira**  
Secretária Executiva de Cooperação com os Municípios

**Helder Nogueira Andrade**  
Secretário Executivo de Equidade, Direitos Humanos e Educação Complementar  
e Protagonismo Estudantil

**Maria Jucineide da Costa Fernandes**  
Secretária Executiva de Ensino Médio e Profissional

**Maria Oderlânia Torquato Leite**  
Secretária Executiva de Gestão da Rede Escolar

**Stella Cavalcante**  
Secretária Executiva de Planejamento e Gestão Interna

**Julianna da Silva Sampaio**  
Coordenadora de Comunicação

**Marta Emilia Silva Vieira**  
**Danielle Taumaturgo Dias Soares**  
**Keifer Fortunatti**  
Assessoras Especiais do Gabinete

**Ideigiane Tercerito Nobre**  
Coordenadora de Gestão Pedagógica do Ensino Médio

**Maria da Conceição Alexandre Souza**  
Articuladora de Gestão

**Dóris Sandra Silva Leão**  
Orientadora da Célula de Gestão Pedagógica e Desenvolvimento Curricular – CEGED

**Francisco Clerto Alves da Silva**  
Orientador da Célula da Educação de Jovens e Adultos e Ensino Médio Noturno – CEJEN

## **Coordenação**

Centro de Documentação e Informações Educacionais  
Coordenadoria de Gestão Pedagógica do Ensino Médio - COGEM

## **Conselho Editorial**

Adriana Schneider Muller Konzen	Izabelle de Vasconcelos Costa
Ana Gardennya Linard Sírío Oliveira	Jacqueline Rodrigues Moraes
Ana Joza de Lima	José Romário Rodrigues Bastos
Antônia Varele Gama Silva	Katiany do Vale Abreu
Antonio Helonis Borges Brandão	Lindalva Costa Cruz
Arnaldo Dias Ferreira	Marco Aurélio Jarreta Merichelli
Augusto Ridson de Araújo Miranda	Marcos Felipe Vicente
Betânia Maria Gomes Raquel	Maria de Fátima Xavier
Cintia Ferreira de Andrade	Mayara Tâmea Santos Soares
Cintya Kelly Barroso Oliveira	Newton Malveira Freire
Elaine Holanda Maciel	Paula de Carvalho Ferreira
Fernanda Maria Diniz da Silva	Paulo Venício Braga de Paula
Francisca Aparecida Prado Pinto	Renata Priscila Conceição da Costa
Francisca Juliana Feitosa Soares	Roberta Eliane Gadelha Aleixo
Francisco de Assis Sales e Costa Junior	Ronaldo Glauber Maia de Oliveira
Francisco Felipe de Aguiar Pinheiro	Rosendo Freitas de Amorim
Gezenira Rodrigues da Silva	Tamara da Cunha Gonçalves
Helayne Mikaele Silva Lima	Vagna Brito de Lima
Herman Wagner de Freitas Regis	Yure Pereira de Abreu

## **Edição**

Prof. Me. Paulo Venício Braga de Paula  
Prof. Dr. Antonio Helonis Borges Brandão  
Centro de Documentação e Informações Educacionais

## **Normalização Bibliográfica**

Elizabete de Oliveira da Silva

## **POLÍTICA EDUCACIONAL E PRODUÇÃO TEXTUAL**

A sociedade brasileira precisa reconhecer efetivamente a relevância da Educação. Um aspecto central desse reconhecimento reside em valorizar o Magistério e o professor. A valorização do magistério pode expressar-se por meio de várias funções e ações desenvolvidas pelo professor. Em 2008, foi instituída uma política pública de estado denominada Professor Aprendiz, cujo destaque tem sido a formação contínua entre pares. A consolidação dessa proposta que investe no protagonismo docente gerou desdobramentos substanciais, dentre os quais se destaca a publicação de livros de professores da rede. Os trabalhos acadêmicos e literários, selecionados para publicação, passam por um criterioso processo de seleção.

A decisão da Secretaria da Educação do Estado do Ceará (Seduc), em organizar e publicar artigos que são recortes de dissertação e tese de professores da rede estadual de ensino, está baseada no programa Ceará Educa Mais, através da ação Professor Aprendiz, do Programa Aprender pra Valer. Esse Programa tem como principais objetivos: a) Valorizar os professores por meio da publicação das suas produções acadêmicas e literárias; b) Estimular a produção científica e literária de professores; c) Promover uma rede de colaboração entre os professores ao tornar públicas suas produções com seus pares.

Com essa iniciativa, a Secretaria da Educação do Estado do Ceará tem feito história. Ao publicar as produções de seus professores, a Seduc tem promovido um círculo virtuoso de valorização do Magistério, cujos efeitos têm se manifestado na consolidação do protagonismo docente; no investimento da formação acadêmica e, principalmente, num processo de ensino e aprendizagem com mais qualidade e compromisso.

**Eliana Nunes Estrela**  
Secretária da Educação do Ceará

**Jucineide Fernandes**  
Secretária Executiva do Ensino Médio e da Educação Profissional

## **PUBLICAÇÃO DAS PRODUÇÕES ACADÊMICAS E LITERÁRIAS DOS PROFESSORES DA REDE PÚBLICA ESTADUAL DE ENSINO DO ESTADO DO CEARÁ**

Existem múltiplas formas de valorização da Educação, uma delas consiste em valorizar a/o professora/or. O reconhecimento da atividade do magistério pode manifestar-se por meio de várias funções e ações desenvolvidas pela/o professora/or.

Em 2008, foi criada uma ação governamental denominada Professor Aprendiz, cujo destaque tem sido a formação continuada por pares. O amadurecimento dessa ação ocorre com a edição da Lei nº 17.572/2021, de 22 de julho de 2021, que estabelece o Programa “Ceará Educa Mais” e que, no Art. 2º, Inciso II, trata da ação Professor Aprendiz. Este programa aposta no protagonismo docente gerando desdobramentos substanciais, dentre os quais destaca-se a publicação de livros de professores(as) da rede que ocorreu nos anos de 2017, 2018 e 2019. Deve ser ressaltado que os trabalhos acadêmicos, literários e temáticos selecionados para publicação passam por um rigoroso processo público de submissão.

A iniciativa da Secretaria da Educação do estado do Ceará (Seduc) em publicar livros produzidos pelos professores da rede estadual de ensino está baseada na ação Professor Aprendiz, do Programa Aprender pra Valer, tendo como principais objetivos: a) a publicação de suas experiências e reflexões; b) a formação e o desenvolvimento contínuo de outros professores; c) na publicação de obras acadêmicas e literárias dos professores, em formato impresso, bem como de livros temáticos, em formato digital.

As obras publicadas podem ser de natureza acadêmica (Tese de Doutorado ou Dissertação de Mestrado), Literária (Romance; Poema; Cordel; Novela; Crônica ou Conto) e Livros Temáticos Digitais que contemplem temas transversais e/ou associados às áreas de conhecimento (Ciências Humanas e Sociais Aplicadas, Linguagem e suas tecnologias, Matemática, Ciências da Natureza e suas tecnologias).

São produções de professores(as) da rede pública estadual de ensino do Ceará, na condição de autor(es) ou coautor(es) da(s) obra(s). O Conselho Editorial, ao selecionar as produções acadêmicas, considerou: clareza e precisão de conteúdo; relevância e atualidade do tema; originalidade; qualidade metodológica. Em relação às produções literárias,



observou-se os seguintes aspectos: originalidade de conteúdo/ineditismo; repertório linguístico; fruição estética; coerência e consistência do texto; e, por último, potencial artístico. Os trabalhos publicados são originais, escritos em língua portuguesa em consonância com os Direitos Humanos.

A Secretaria da Educação do Estado do Ceará mais uma vez faz história com essa iniciativa. Ao publicar as produções de seus(suas) professores(as), a Seduc promove um círculo virtuoso de valorização do Magistério, cujos efeitos podem se manifestar no fortalecimento do protagonismo docente; no investimento da formação acadêmica e, principalmente, num processo de ensino e aprendizagem mais qualificado e comprometido.

**Prof. Dr. Antonio Helonis Borges Brandão**  
**Prof. Dr. Rosendo Freitas de Amorim**  
**Prof. Ms. Paulo Venício Braga de Paula**

## PRÓLOGO

Rubem Alves, sobre os olhos, as palavras e o mundo, assim reflete: “As palavras só têm sentido se nos ajudam a ver o mundo melhor. Aprendemos palavras para melhorar os olhos”. Dada a luz poética com que esse grande educador costumava iluminar seus pensamentos em forma de discurso, dificilmente os sentidos possíveis para esses três signos se esgotam. Os olhos podem ser muito mais do que as córneas, as palavras podem ir muito além dos verbetes e podemos conhecer vários mundos, como o mundo das/os educadoras/es.

Nesse mundo, os olhos representam toda a sensibilidade do indivíduo que educa. A/o educadora/or vê não só com os olhos, mas também com os ouvidos e com o tato. Tudo, ao seu redor, é palavra: críticas e elogios, respostas “certas” e “erradas”, perguntas e silêncios, abraços e distâncias, sorrisos e lágrimas. Entretanto, como educadoras/es, nem sempre nos damos conta de respirar tantos significados nessa semiosfera que é a escola e podemos, muitas vezes, ignorá-los. E assim, perdemos a oportunidade de melhorar nossos olhos.

Esta publicação traz valiosas contribuições de educadoras e educadores que aproveitam essa oportunidade e, agora, também nos oportunizam uma melhora do nosso modo de ver a educação. As produções aqui apresentadas trazem a perspectiva de quem aprimorou um olhar pedagógico que, agora, transforma em palavra.

A Secretaria da Educação do Estado do Ceará (Seduc), por meio da Coordenadoria de Gestão Pedagógica do Ensino Médio, espera que as palavras das/os nossas/os educadoras/es, aqui eternizadas, possam alcançar (e melhorar) os mais diversos olhares. Que esses olhares possam germinar em produções futuras que contribuirão, cada vez mais, com o nosso modo de compreender e de agir neste mundo tão desafiador, que é o da educação.

**Ideigiane Terceiro Nobre**

Coordenadora da Gestão Pedagógica do Ensino Médio/COGEM

**Ana Cecília Freitas**

Assistente Educacional /COGEM



*À nossa filha, Maria.*





## SOBRE OS AUTORES

Adeil Araújo é professor de Física na rede pública estadual do Ceará. É mestre em Ensino de Física e graduado em Licenciatura em Física pela Universidade Estadual do Ceará. É autor do produto educacional Física em Movimento: Atividades com Smartphones e Tablets. Tem como hobby estudar. Adora aprender coisas novas e é um apaixonado por compartilhar o que aprende.

Meirivâni Meneses é professora de Matemática na rede pública estadual do Ceará. É mestre em Ensino de Ciências e Matemática pela Universidade Federal do Ceará e graduada em Licenciatura em Matemática pela Universidade Estadual do Ceará. É autora do produto Educacional Smartmática: A Matemática do dia a dia através da Videoanálise.





# AGRADECIMENTOS

Várias pessoas tiveram participação na escrita deste livro. Corremos o risco de cometer alguma injustiça, mas bora lá!

Aos nossos pais, pelos conselhos e apoio durante toda a nossa caminhada na educação.

À nossa filha Maria, que sempre de manhã cedinho vai à nossa cama avisar que o Sol já nasceu e que é hora de brincar.

Aos nossos familiares, que por várias vezes cuidaram da nossa filha para que pudéssemos desenvolver nossos projetos.

Ao Adailton Araújo e ao Lucas Queiroz, pela leitura e contribuições durante a escrita deste livro.

À professora Dra. Eloneid Felipe Nobre, pelo incentivo e orientação durante nossa formação.

Às/os nossas/os colegas de trabalho, alunas/os e ex-alunas/os, pois estamos diariamente aprendendo com elas/es.

À Secretaria da Educação do Estado do Ceará, pelas políticas de valorização do trabalho dos professores.



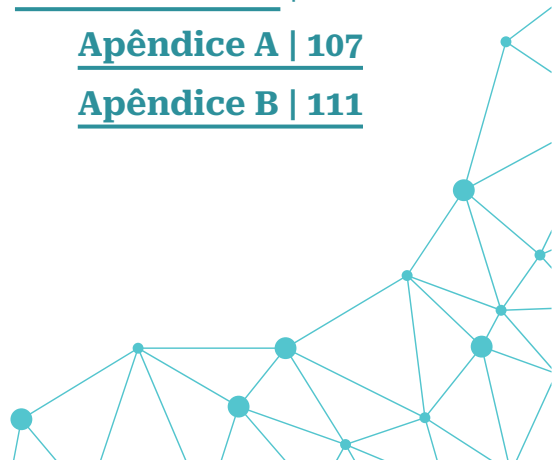




# SUMÁRIO

Sobre os autores	11
Agradecimentos	13
Sumário	15
Bora lá!	17
<b><u>Capítulo 01</u></b> - Input	19
<b>1.1</b> Um pouco de história	19
<b>1.2</b> Conhecendo os possíveis ambientes de desenvolvimento	20
<b>1.2.1</b> Unindo o Julia ao Jupyter	22
<b><u>Capítulo 02</u></b> - Variáveis	27
<b>2.1</b> Codificando um problema de lançamento vertical	29
<b>2.2</b> Formatando números e strings	32
<b>2.3</b> Notação Científica	33
<b><u>Capítulo 03</u></b> - Pacotes	35
<b>3.1</b> Instalação de pacotes	35
<b>3.2</b> Pacote Unitful	36
<b>3.2.1</b> Adicionando unidades e realizando conversões	37
<b><u>Capítulo 04</u></b> - Gráficos	41

4.1 Construindo um gráfico   42	<b><u>Capítulo 07 -</u></b> Broadcasting   75
4.2 Resolvendo o problema do encontro de móveis com o Plots   46	7.1 Broadcasting em funções trigonométricas   79
4.3 Arrays   48	7.1.1 Gráficos do Movimento Harmônico Simples   81
4.3.1 Acompanhando a perda de peso   50	<b><u>Capítulo 08 -</u></b> Decisão   85
<b><u>Capítulo 05 -</u></b> Importando e Exportando Dados   53	8.1 if/elseif/else   86
5.1 Pacote CSV   54	8.2 Operador ternário   88
5.2 Construindo gráficos com dados de um arquivo CSV   57	8.3 Operadores E (&&) e OU (  )   89
5.3 Pacote DataFrames   60	<b><u>Capítulo 09 -</u></b> Repetição   91
<b><u>Capítulo 06 -</u></b> Funções   63	9.1 Laço for   91
6.1 Calculando o peso e inserindo as unidades de medidas   66	9.2 Laço <i>while</i>   95
6.2 Calculando o Trabalho de uma força   68	<b><u>OUTPUT</u></b>   101
6.3 Argumentos   70	<b><u>BIBLIOGRAFIA</u></b>   103
	<b><u>Apêndice A</u></b>   107
	<b><u>Apêndice B</u></b>   111





# BORA LÁ!

Já se passaram quase 15 anos desde que começamos a trabalhar nas escolas públicas estaduais do Ceará. Quem nos conhece sabe que, durante todo esse período, buscamos diversificar nossa prática pedagógica por meio de experimentos com materiais de baixo custo e com as Tecnologias Digitais de Informação e Comunicação, como aplicativos de videoanálise, a linguagem Python e outras linguagens exploradas nos cursos da Code.org.

Elaboramos, então, este livro, com o objetivo de contribuir com a inserção de uma linguagem de programação nas escolas de Ensino Médio. Ele apresenta a linguagem Julia, através de exemplos simples de Física básica.

## **Por que escolhemos a linguagem de programação Julia?**

Nossa escolha por Julia deve-se ao fato dela ser uma

linguagem de sintaxe fácil, quando comparada com outras linguagens de programação, além de permitir escrever fórmulas matemáticas da mesma forma como aprendemos na escola, o que facilita sua introdução no Ensino Médio.

### **Para quem este livro é destinado?**

Este livro tem como público-alvo professoras/es e alunas/os do Ensino Médio. No entanto, qualquer pessoa que queira aprender Julia pode aventurar-se por ele.

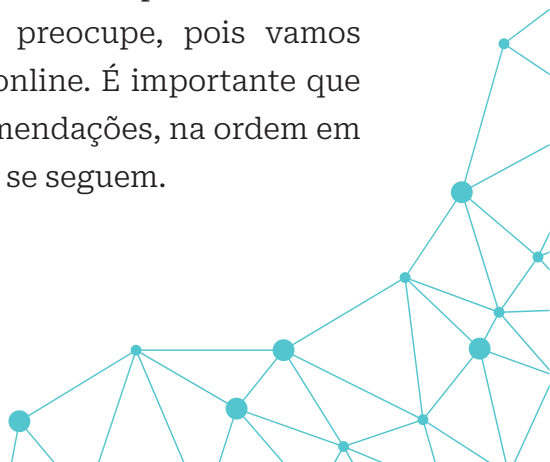
### **O que este livro apresenta?**

Este livro apresenta conceitos e princípios básicos de codificação. Nele, ensinamos a resolver problemas simples, a plotar gráficos, inserir unidades de medidas nos resultados, criar funções, importar e exportar dados e a trabalhar com controle de fluxo. Esta edição do livro está baseada na versão 1.5.1 de Julia lançada em agosto de 2020.

### **Como usar este livro?**

Aconselhamos que você leia este livro reproduzindo os códigos em um computador. Não se preocupe, pois vamos ensinar a instalar o Julia e a utilizá-lo online. É importante que você experimente todas as nossas recomendações, na ordem em que são apresentadas nos capítulos que se seguem.

E agora... Bora lá!

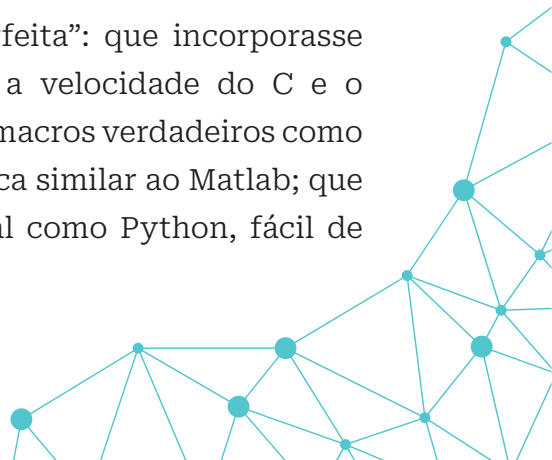




## 1.1 Um pouco de história

A linguagem de programação Julia foi desenvolvida em 2009 por Jeff Bezanson, Alan Edelman e Stefan Karpinski. Viral B. Shah e teve sua primeira versão de código aberto lançada em 2012. Segundo os próprios autores, todos eles são usuários de Matlab, Python, Ruby, Lisp, R, C e amam todas essas linguagens. No entanto, cada uma delas é perfeita em alguns aspectos, mas falham em outros.

Insatisfeitos com algumas características de cada uma dessas linguagens, eles resolveram unir seus conhecimentos e criar uma linguagem que fosse “perfeita”: que incorporasse o código aberto com licença liberal, a velocidade do C e o dinamismo do Ruby; que contasse com macros verdadeiros como a do Lisp e com uma notação matemática similar ao Matlab; que fosse utilizável para programação geral como Python, fácil de



usar para estatísticas como R, natural para processamento de strings como Perl, poderosa para álgebra linear como Matlab, boa para juntar programas como o shell e que fosse compilada. Em resumo, uma linguagem extremamente fácil de aprender e que apresentasse todas essas vantagens.

Julia é, portanto, uma linguagem muito poderosa para computação científica, aprendizado de máquina, mineração de dados, álgebra linear em larga escala, computação distribuída e paralela. Para entendermos melhor, na próxima seção iremos conhecer alguns ambientes de desenvolvimento.

## **1.2 Conhecendo os possíveis ambientes de desenvolvimento**

Se você é principiante neste universo da programação, talvez não se sinta confortável para instalar o Julia em seu computador, mas não se preocupe: é possível rodar o Julia em seu navegador. Existem diversas plataformas que oferecem um local para desenvolvimento e execução de programas online, como por exemplo o repl.it<sup>1</sup>. Essa plataforma oferece suporte ao desenvolvimento em mais de 50 linguagens de programação, como Python, C, C++, Java, HTML, CSS, JS, Ruby, Go e Julia. Para ter acesso ao ambiente repl.it e criar seus primeiros programas, basta cadastrar-se usando seu Gmail ou Facebook. A interface do editor do repl.it é apresentada na Figura 1.1.

1. <https://repl.it/>

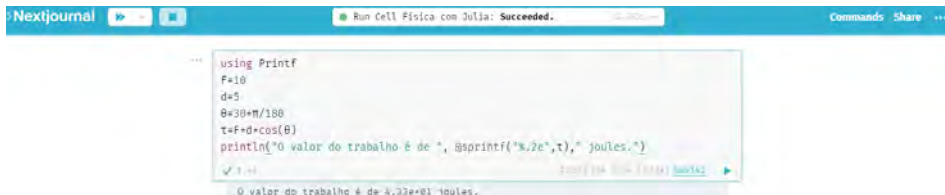
Figura 1.1 – Captura da tela do editor do repl.it.



Fonte: Elaborada pelos autores.

Outra opção para rodar o Julia em seu navegador é por meio do Nextjournal.<sup>2</sup> Aqui, você também terá que criar uma conta usando o seu Gmail. Porém, o ambiente de desenvolvimento dessa plataforma é bem diferente do anterior. Enquanto o repl.it aceita apenas códigos, o nextjournal é um ambiente computacional destinado à criação de documentos que podem conter, além de códigos, textos, imagens, vídeos e equações em um só lugar. Esse tipo de ambiente é chamado de **notebook**. É um dos nossos preferidos porque além de ser bem completo, apresenta a vantagem de ter o suporte para eventuais erros. Sempre que algo dá errado com seu programa, aparece a opção HELP e você pode usá-la para acionar algum colaborador. Além do Julia, essa plataforma suporta as linguagens Python e R. Sua interface pode ser vista na Figura 1.2.

Figura 1.2 – Captura de tela do editor do nexjournal.com.



Fonte: Elaborada pelos autores.

2. <https://nextjournal.com>

Outra opção de *notebook* é o jupyter<sup>3</sup> (acrônimo para Julia, Python e R). Você tanto pode usá-lo em seu navegador como fazer o *download* em seu computador. A interface de seu site pode ser vista na Figura 1.3 e, no apêndice A, você encontrará instruções que te ajudarão a acessar o Jupyter<sup>4</sup> **online** pela primeira vez.

Figura 1.3 – Captura de tela do editor do site do jupyter.org.



Fonte: Elaborada pelos autores.

Na próxima seção, apresentaremos os passos para unir Julia ao Jupyter no computador (caso você queira, claro!). Se não desejar instalar nada, você estará pronto para programar. Vá para o capítulo 2, escolha sua plataforma preferida e bons estudos!

### 1.2.1 Unindo o Julia ao Jupyter

Para unir o Julia ao Jupyter, primeiramente instale o Julia em seu computador. Para realizar esta operação você deverá ir até o site do [julialang.org](https://julialang.org)<sup>5</sup> e escolher dentre as opções de instalação:

3. <https://jupyter.org/>

4. Durante a escrita deste livro, optamos por utilizar o notebook jupyter instalado em nosso computador para não ficarmos dependentes da internet.

5. <https://julialang.org/downloads/>



Windows, MacOS, e Linux, a que melhor se adequa ao seu sistema operacional. No nosso caso, instalamos o Julia para Windows 64 bits. A captura de tela do site com as opções para os sistemas operacionais citados pode ser vista na Figura 1.4.

Figura 1.4 – Captura de tela do site do [julialang.org/downloads/](https://julialang.org/downloads/).

### Current stable release: v1.5.1 (Aug 25, 2020)

Checksums for this release are available in both [MD5](#) and [SHA256](#) formats.

<b>Windows</b> <a href="#">[help]</a>	64-bit (installer), 64-bit (portable)		32-bit (installer), 32-bit (portable)
<b>macOS</b> <a href="#">[help]</a>	64-bit		
<b>Generic Linux on x86</b> <a href="#">[help]</a>	64-bit (GPG), 64-bit (musl) <sup>[1]</sup> (GPG)		32-bit (GPG)
<b>Generic Linux on ARM</b> <a href="#">[help]</a>	64-bit (AArch64) (GPG)		
<b>Generic FreeBSD on x86</b> <a href="#">[help]</a>	64-bit (GPG)		
<b>Source</b>	Tarball (GPG)	Tarball with dependencies (GPG)	GitHub

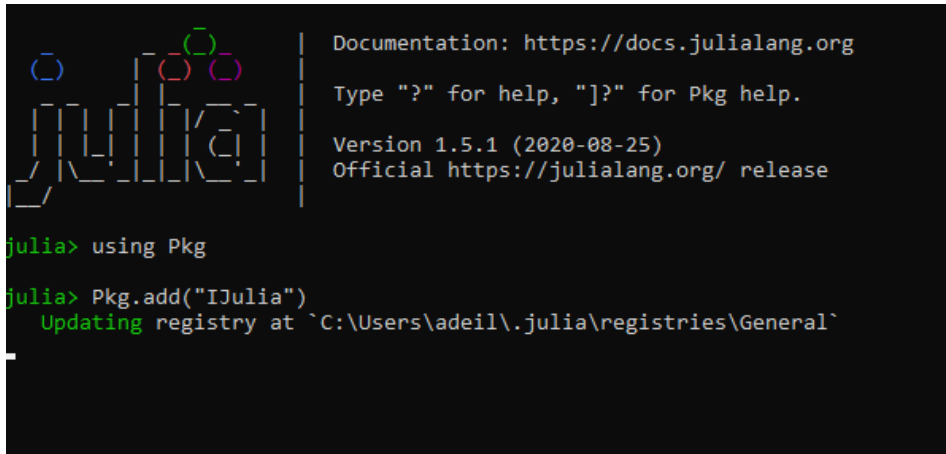
Fonte: Elaborada pelos autores.

Após a instalação, ao abrir o Julia, você verá uma tela preta com a logo da linguagem e o nome `julia>_`,<sup>6</sup> com o símbolo *underline* piscando. É aí que você deverá digitar. Esse ambiente é conhecido como REPL (*read-eval-print-loop*) e ele por si só é suficiente para iniciarmos, mas como queremos unir o Julia ao Jupyter, vamos prosseguir.

O próximo passo é digitar o comando **using Pkg**, pressionar [ENTER] e, em seguida, digitar **Pkg.add("IJulia")** e novamente pressionar [ENTER]. Isso poderá demorar alguns minutinhos. Essas instruções podem ser visualizadas na Figura 1.5.

6. Caso esteja utilizando um MacOS a tela será branca e não aparecerá o símbolo `_` piscando.

Figura 1.5 – Captura de tela linha da linha de comando do Julia (REPL).



```
Documentation: https://docs.julialang.org
Type "?" for help, "]" for Pkg help.
Version 1.5.1 (2020-08-25)
Official https://julialang.org/ release

julia> using Pkg

julia> Pkg.add("IJulia")
Updating registry at `C:\Users\adeil\.julia\registries\General`
```

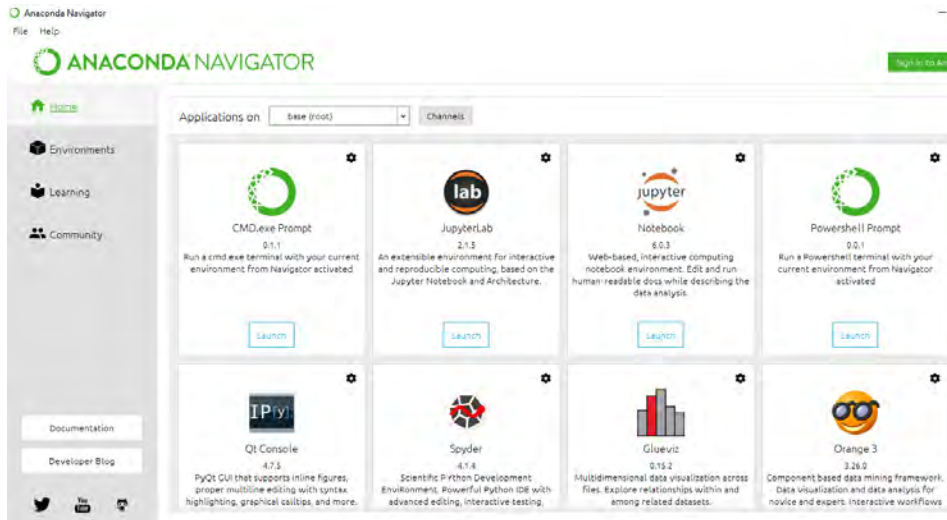
Fonte: Elaborada pelos autores.

Digitar esses comandos fará com que o Jupyter reconheça automaticamente o Julia em seu computador quando ele for instalado.

Agora, para ter acesso ao Jupyter, você deverá abrir uma nova aba em seu navegador e instalar o Anaconda.<sup>7</sup> O Anaconda é uma plataforma para ciência de dados que permite a programação em diversas linguagens. Uma vez instalado, basta clicar em *Launch* na opção jupyter notebook, como mostra a Figura 1.6.

7. <https://www.anaconda.com/products/individual>

Figura 1.6 – Captura de tela da interface do Anaconda.



Fonte: Elaborada pelos autores.

Para criar um novo **notebook** clique em **New**, no canto superior direito da tela, e escolha a opção Julia, que você instalou em seu computador. Veja a Figura 1.7.

Figura 1.7 – Captura de tela do painel de controle (**dashboard**) do jupyter notebook.



Fonte: Elaborada pelos autores.

Pronto! O próximo passo é digitar seu primeiro código.





Quando estamos resolvendo um problema de Física, por exemplo, costumamos retirar os dados da questão e atribuir esses valores a letras para identificarmos mais facilmente no momento que precisarmos. Em programação, fazemos o mesmo. Quando queremos guardar uma informação, damos a ela um nome e a chamamos de variável. Pode-se dizer, então, que uma variável é uma caixa que contém algum tipo de “dado”.

As variáveis podem ser escritas por meio de letras e números em qualquer ordem, sem espaço, mas não podem iniciar com um número. Outra informação que você precisa saber é que o valor que é atribuído a uma variável define o seu tipo e isso é feito automaticamente em Julia. Vejamos alguns exemplos de variáveis na Figura 2.1.

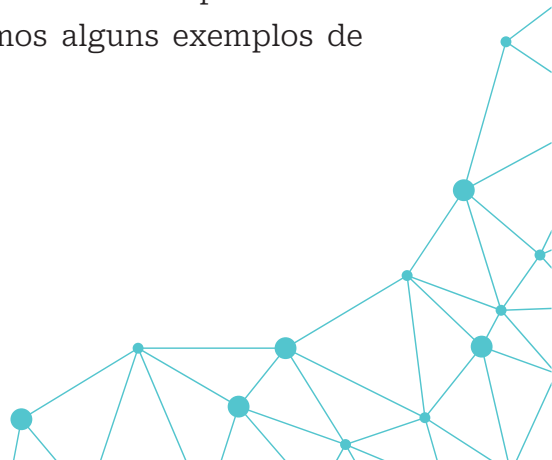


Figura 2.1 – Tipos de variáveis em Julia.

```
In [1]: v0=7 # m/s
        typeof(v0)
Out[1]: Int64

In [3]: g=9.81 #m/s^2
        typeof(g)
Out[3]: Float64

In [2]: t=0.4 #s
        typeof(t)
Out[2]: Float64

In [3]: z="z"
        typeof(z)
Out[3]: String

In [4]: x="Eu gosto de Física e de Julia"
        typeof(x)
Out[4]: String

In [5]: a='3'
        typeof(a)
Out[5]: Char
```

Fonte: Elaborada pelos autores.

Você deve ter observado na figura que o ambiente do Jupyter é dividido em células. Tudo o que digitamos está na célula **In [ ]:** e tudo o que o programa nos retorna está em **Out [ ]:**. Na primeira célula, atribuímos à variável  $v_0$  o valor de 7. Para sabermos de que tipo é essa variável, digitamos **typeof( $v_0$ )**, que em português significa “tipo de”. Ao realizar essa ação, o programa nos retornará automaticamente, Int64. Isso indica que esse valor é um inteiro de 64 bits. No momento não iremos entrar em detalhes, o importante é você entender que 7 é um número inteiro.

No segundo exemplo, atribuímos à variável  $g$  o valor de 9.81 e em seguida digitamos **typeof( $g$ )** para sabermos o seu tipo. Veja que o programa nos retornou um Float64. Isso significa que  $g$  é um número real, que em computação chamamos de números de ponto flutuante ou simplesmente *float*.

Nos dois próximos exemplos, digitamos, respectivamente,  $z = \text{"z"}$  e  $x = \text{"Eu gosto de Física e de Julia"}$ . Em ambas as situações, o programa nos retornou `String`. `Strings` são textos que podem conter qualquer caractere Unicode, como por exemplo:  $\alpha$ ,  $\beta$ ,  $\gamma$  e  $\theta$ . Mais adiante, veremos como escrever esses símbolos em Julia.

No último exemplo, digitamos  $a = \text{'3'}$ , com 3 entre aspas simples. Ao digitar em seguida `typeof(a)`, o programa nos retornou **Char** (caractere), que é um outro tipo de variável em Julia. Existem outros tipos de variáveis, mas abordaremos as que mais nos interessam neste momento, que são: `Int64`, `Float64` e as `Strings`.

Pronto, agora que sabemos o que é uma variável e conhecemos alguns de seus tipos, podemos ir ao nosso primeiro exemplo que será apresentado na próxima seção.

## 2.1 Codificando um problema de lançamento vertical

Suponha que uma bola é lançada, verticalmente para cima, em um local onde se possa desprezar a resistência do ar e próximo à superfície da Terra. O modelo matemático que prevê a posição vertical  $y$  da bola em função do tempo  $t$  é dado pela equação 2.1.

$$y = v_0 \cdot t - 0,5 \cdot g \cdot t^2 \quad (2.1)$$

Onde  $v_0$  é a velocidade inicial da bola e  $g$  é a aceleração da gravidade, cujo valor é algo em torno de  $9.81 \text{ m/s}^2$ .

De posse desta equação, e com uma velocidade inicial

conhecida, você poderá inserir um valor para o tempo e obter a posição vertical (altura) correspondente. Vejamos a seguir, na Figura 2.2, um programa no qual atribuímos a  $v_0$  o valor 7 m/s e a  $t$ , o valor 0.4 s. Antes de continuarmos, gostaríamos de fazer um adendo: em Julia, o sinal = significa atribuição, enquanto o sinal de == significa igualdade.

Figura 2.2 – Programa para calcular a altura de uma bola em movimento vertical.

```
In [1]: # Programa para calcular a altura de uma bola em movimento vertical
v0 = 7 # velocidade inicial em m/s
g = 9.81 # aceleração da gravidade em m/s^2
t = 0.4 # tempo em segundos
y = v0*t - 0.5*g*t^2 # posição vertical em metros
println(y)

2.0152
```

Fonte: Elaborada pelos autores.

Quando você executar o programa, clicando em **Run** ou ([SHIFT][ENTER], [ALT][ENTER]), seu código será lido linha por linha, de cima para baixo e da esquerda para a direita.

Observe que a primeira linha do nosso código é:

**# Programa para calcular a altura de uma bola em movimento vertical**

Saiba que tudo o que vier após o símbolo da cerquilha “#” é considerado um comentário. Dessa forma, Julia não lê nem executa esta linha, mas ela é considerada uma boa prática em programação, pois nos ajuda a entender nossos códigos. Quando o comentário possui várias linhas, devemos delimitá-lo com #=# para abrir e fechar o comentário, respectivamente.



Na segunda linha, atribuímos à variável  $v_0$  o valor de 7 m/s. Para que Julia escreva  $v_0$  em seu código, devemos digitar `v_0` e em seguida, pressionarmos a tecla [TAB]. Como vimos,  $v_0$  é do tipo inteiro (Int64).

Na terceira e quarta linha, atribuímos a  $g$  o valor de  $9.81 \text{ m/s}^2$  e a  $t$ , o valor 0.4 s. Perceba que ambos são números reais, do tipo Float64. Estamos lembrando a vocês de que tipos são essas variáveis, apenas para exercitarmos o que já aprendemos até aqui.

Agora que atribuímos valores as variáveis  $v_0$ ,  $g$  e  $t$ , devemos digitar a equação:  $y = v_0 \cdot t - 0,5 \cdot g \cdot t^2$  para que Julia calcule o valor da altura atingida pela bola.

É importante destacar que o símbolo (\*) é interpretado como um sinal de multiplicação, o símbolo do circunflexo (^), como potenciação e os sinais (+), (-), e (/) são utilizados para as operações de adição, subtração e divisão, respectivamente.

Por fim, na última linha, insira a função **println(y)** para que Julia possa lê e imprimir na tela o valor de  $y$ , que neste caso é 2.0152. Em nossos resultados, o separador decimal será um ponto e não uma vírgula, como estamos acostumados.

Caso queira ter várias variáveis em uma mesma linha, basta separá-las por ponto e vírgula. A desvantagem de se declarar as variáveis dessa forma é que se torna difícil comentar o código, por isso, este tipo de declaração deve ser evitado. Veja um exemplo na Figura 2.3.

Figura 2.3 – Programa com várias declarações na mesma linha separadas por ponto e vírgula.

```
In [4]: # Programa para calcular a altura de uma bola em movimento vertical
v0 = 7; g = 9.81; t = 0.4
y = v0*t - 0.5*g*t^2
println(y)

2.0152
```

Fonte: Elaborada pelos autores.

## 2.2 Formatando números e strings

No exemplo da seção anterior, o resultado gerado possui quatro casas decimais. É necessária tanta precisão? E se quiséssemos informar o resultado com duas casas decimais, como deveríamos proceder? Na Figura 2.4 apresentaremos as alterações necessários para realizar esta ação.

Figura 2.4 – Formatação de string usando a macro @sprintf.

```
In [7]: # Programa para calcular a altura de uma bola em movimento vertical
using Printf
v0 = 7 # velocidade inicial
g = 9.81 # aceleração da gravidade
t = 0.4 # tempo
y = v0*t - 0.5*g*t^2 # posição vertical
println("O valor de y é de aproximadamente ", @sprintf("%.2f",y)," metros.")

O valor de y é de aproximadamente 2.02 metros.
```

Fonte: Elaborada pelos autores.

Para começar, na segunda linha do nosso programa importamos o pacote **Printf**. Ele possui a macro @sprintf que vamos utilizar. Macros são como funções na Matemática. Elas recebem expressões de entrada e as retornam modificadas.

Na última linha, observe que a macro @sprintf possui o termo (“%.2f”, y). Esse termo faz com que o valor de y tenha apenas 2 casas decimais, como mostra a saída do código.

Na próxima seção veremos outra aplicação desta macro.

## 2.3 Notação Científica

Em muitos problemas, precisamos obter os resultados de nossos cálculos em notação científica. Vamos utilizar o seguinte exemplo para explicar como obter esta formatação.

Sabemos que a distância entre o Sol e a Terra é de aproximadamente 150000000 km (150 milhões de km). O código na Figura 2.5 apresentará este número em notação científica.

Figura 2.5 – Distância do Sol à Terra em notação científica.

```
In [8]: #Usando o pacote Printf e a macro @sprintf para colocar um número em notação científica
using Printf
distancia=150000000 #distância do Sol à Terra em quilômetros.
println("A distância entre o Sol e o planeta Terra é de aproximadamente ",@sprintf("%.1e",distancia), " de quilômetros.")
A distância entre o Sol e o planeta Terra é de aproximadamente 1.5e+08 de quilômetros.
```

Fonte: Elaborada pelos autores.

Vamos entender as linhas desse código.

1. A primeira linha do código, como já sabemos, é um comentário. Julia não o lerá;
2. Na linha seguinte, importamos o pacote **Printf**;
3. Na terceira linha, criamos a variável “distancia” e atribuímos o valor de 150 milhões de quilômetros;
4. Na última linha inserimos a função **println()** e dentro dela, a macro **@sprintf("%.1e”,distancia)**. O termo (“%.1e”,distancia) formata o valor da distância para notação científica com uma casa decimal e o resultado é o número no formato (1.5e+08), que é o mesmo que  $1,5 \cdot 10^8$ .

Os programas que discutimos podem ser testados com outros valores. Vá em frente!





Julia possui uma comunidade *online* bastante ativa<sup>1</sup> que desenvolve inúmeros pacotes para o seu ecossistema. Existem pacotes com as mais diversas funcionalidades e que atendem às mais variadas áreas, como: Química, Matemática, Finanças, Estatística, Cosmologia, Linguística, dentre outras. Atualmente, existem mais de 4 mil pacotes registrados. Você pode consultar o site [julialang.org](https://julialang.org)<sup>2</sup> e descobrir um que atenda às suas necessidades.

Neste capítulo, explicaremos como instalar um pacote e apresentaremos o pacote Unitful.<sup>3</sup> Este pacote é útil para inserir unidades de medidas e realizar conversões.

### 3.1 Instalação de pacotes

Para instalar um pacote, você precisará inserir os seguintes comandos:

1. <https://julialang.org/community/>
2. <https://julialang.org/packages/>
3. <https://juliapackages.com/p/unitful>



1. `using Pkg;`
2. `Pkg.add("NOME DO PACOTE").`

Se você estiver utilizando, por exemplo, o Jupyter notebook instalado em sua máquina, esse procedimento é necessário uma única vez. Caso esteja usando o notebook Jupyter *online*, esses comandos devem ser digitados toda vez que você iniciar uma nova seção.

Após a instalação, para carregar um pacote, basta digitar a palavra-chave *using* seguida do nome do pacote da seguinte forma:

1. `using NOME DO PACOTE.`

Esse comando deverá ser digitado toda vez que você abrir um novo *notebook* e precisar carregar um determinado pacote.

Na próxima seção, ensinaremos como instalar o pacote Unitful. Lançaremos alguns exemplos para ilustrar como podemos incluir unidades de medidas em nossos resultados e fazermos conversões de unidades.

### **3.2 Pacote Unitful**

Em Física estamos sempre trabalhando com unidades de medidas e conversões de uma unidade em outra. Nesta seção, conheceremos o pacote Unitful, que nos ajudará com esta tarefa. Para instalar este pacote, seguimos a mesma orientação dada na seção anterior, como mostra a Figura 3.1.

Figura 3.1 – Instalação do pacote Unitful.

```
In [3]: using Pkg
        Pkg.add("Unitful")

Resolving package versions...
No Changes to `C:\Users\adeil\.julia\environments\v1.5\Project.toml`
No Changes to `C:\Users\adeil\.julia\environments\v1.5\Manifest.toml`
```

Fonte: Elaborada pelos autores.

Na primeira linha, em (In[3]:), usamos o trecho de código *using Pkg* e em seguida, *Pkg.add("Unitful")*. Perceba que, para instalá-lo, apenas trocamos “NOME DO PACOTE” por “Unitful”. Agora é preciso carregá-lo para nosso código, mas isso veremos a seguir.

### 3.2.1 Adicionando unidades e realizando conversões

Para adicionar unidades de medidas aos números, usamos a notação: *u*“unit” ou *\*u*“unit”. Se quisermos escrever 1km, por exemplo, podemos utilizar uma das seguintes sintaxes: *1u*“km” ou *1\*u*“km”. Por sua vez, para fazer uma conversão de unidades, você deverá utilizar o *uconvert*<sup>4</sup>. Vejamos um exemplo na Figura 3.2.

Figura 3.2 – Convertendo quilômetros em metros.

```
In [4]: using Unitful

In [5]: distancia_km=1u"km"

Out[5]: 1 km

In [6]: distancia_m=uconvert(u"m", distancia_km)

Out[6]: 1000 m
```

Fonte: Elaborada pelos autores.

4. <https://painterqubits.github.io/Unitful.jl/stable/conversion/>

Vamos explicar o que inserimos em cada célula:

1. Em (In[4]:) carregamos o pacote Unitful usando a palavra-chave *using*;

2. Na célula seguinte, criamos a variável **distancia\_km** e atribuímos o valor de **1u"km"**. Ao executarmos o código, o programa nos retornará, em (Out[5]:), **1 km**;

3. Na célula (In[6]:), realizamos a conversão da distância em quilômetros para metros, utilizando o `uconvert`. Perceba, em (Out[6]:), que o programa nos retornou 1000 m para essa conversão.

Caso julgue essa sintaxe um pouco estranha, você pode substituí-la pelo operador de encadeamento de função `|>`. Esse operador tem a função de converter uma unidade em outra equivalente. Na Figura 3.3, apresentamos um exemplo de sua utilização.

Figura 3.3 – Utilização do operador `|>`.

```
In [25]: 1u"km" |> u"m"
```

```
Out[25]: 1000 m
```

Fonte: Elaborada pelos autores.



Esse operador também permite converter graus Celsius em graus Fahrenheit ou vice-versa, como mostra a Figura 3.4.

Figura 3.4 – Conversão de Celsius para Fahrenheit com o operador `|>`.

```
In [22]: 20u"°C" |> u"°F"
```

```
Out[22]: 68//1 °F
```

```
In [23]: 36u"°C" |> u"°F"
```

```
Out[23]: 484//5 °F
```

Fonte: Elaborada pelos autores.

No primeiro exemplo, em (In[22]:), convertemos 20 °C em Fahrenheit. O resultado foi 68//1 °F. Você deve estar se perguntando o que significam essas barras duplas (//). Às vezes, quando se deseja cálculos exatos, usam-se números racionais e sua sintaxe é numerador//denominador, por exemplo, 1//3.

Na célula seguinte, transformamos 36°C em Fahrenheit e o resultado foi 484//5 °C, ou seja, a mesma notação do primeiro exemplo. Caso você esteja curioso/a para saber o resultado dessa conta, basta dividir 484 por 5 e terá como resultado 96.8 °F.

Agora, vamos usar pacote Unitful na resolução do seguinte problema: suponha que um objeto se desloque por 10 m durante 2 s. Qual sua velocidade média durante este percurso? Veja a resolução na Figura 3.5.

Figura 3.5 – Cálculo da velocidade média usando o pacote Unitful.

```
In [23]: #Cálculo da velocidade média
Δx= 10u"m" # para escrever Δ devemos digitar: \Delta[TAB]
Δt=2u"s"   # para escrever Δ devemos digitar: \Delta[TAB]
v=Δx/Δt
println("A velocidade média do objeto é de ", v)
```

A velocidade média do automóvel é de 5.0 m s<sup>-1</sup>

Fonte: Elaborada pelos autores.

1. Na primeira linha do código, criamos a variável  $\Delta x$  e atribuímos o valor de 10 m. Escrevemos 10u“m” com o objetivo de adicionar a unidade de medida **m**, ao número 10;
2. Na segunda linha, atribuímos a variável  $\Delta t$  o valor de 2 s;
3. Na linha seguinte, definimos a velocidade média. Por fim, solicitamos o programa para imprimir o valor dessa velocidade.

Repare que a unidade de medida foi atribuída automaticamente pelo programa. O resultado mostra que a velocidade é de 5.0 m.s<sup>-1</sup>, que equivale a 5 m/s.

No próximo capítulo, aprenderemos a construir gráficos utilizando o pacote Plots.



Há muitos pacotes para gerar gráficos disponíveis em Julia, como por exemplo: Gadfly<sup>1</sup> e Winston<sup>2</sup>. No entanto, neste capítulo apresentaremos o pacote Plots<sup>3</sup> e com ele, aprenderemos a construir gráficos. A Figura 4.1 apresenta os comandos necessários para você instalar esse pacote.

Figura 4.1 – Instalação do pacote Plots.

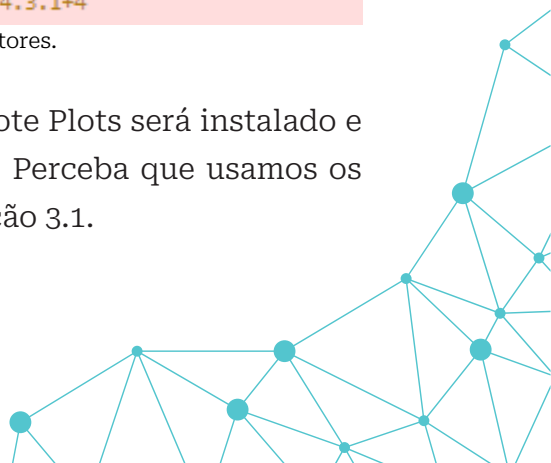
```
In [2]: using Pkg
        Pkg.add("Plots")

Updating registry at `C:\Users\adeil\.julia\registries\General`
Resolving package versions...
Installed FFMPEG_jll - v4.3.1+4
No Changes to `C:\Users\adeil\.julia\environments\v1.5\Project.toml`
Updating `C:\Users\adeil\.julia\environments\v1.5\Manifest.toml`
[b22a6f82] ↑ FFMPEG_jll v4.3.1+2 => v4.3.1+4
```

Fonte: Elaborada pelos autores.

Ao executar a célula (In[2]:), o pacote Plots será instalado e você poderá criar seu primeiro gráfico. Perceba que usamos os mesmos comandos apresentados na seção 3.1.

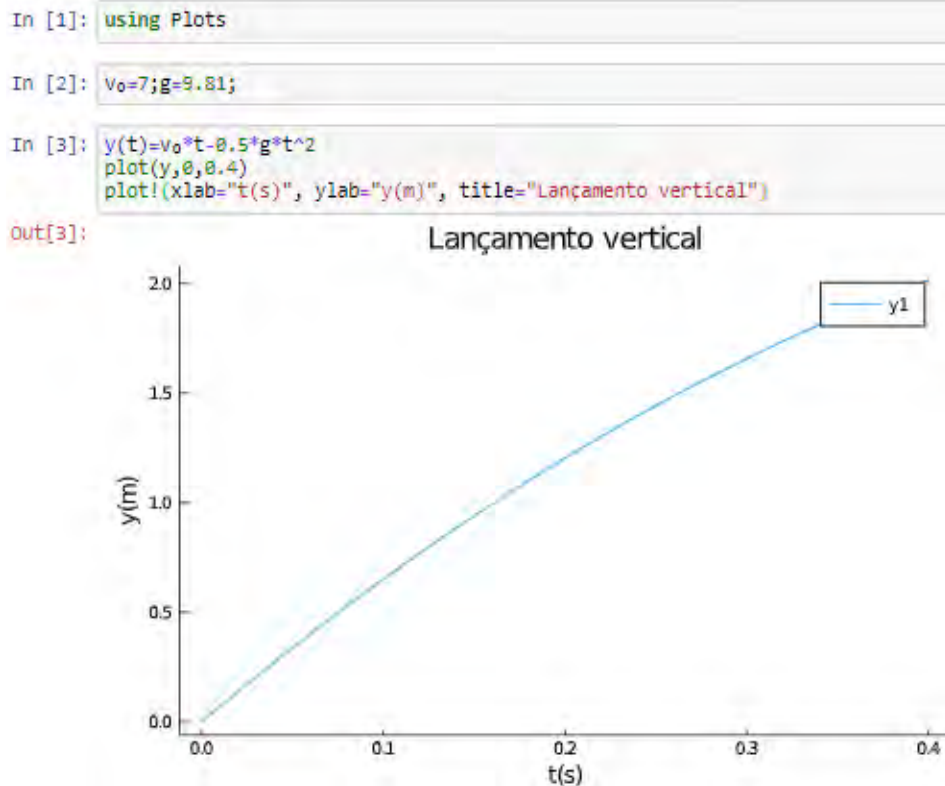
1. <https://github.com/GiovineItalia/Gadfly.jl>
2. <https://github.com/JuliaGraphics/Winston.jl>
3. <https://docs.juliaplots.org/latest/>



## 4.1 Construindo um gráfico

Nesta seção, apresentaremos os comandos necessários para construir gráficos usando o pacote Plots. Utilizaremos o problema do movimento vertical da bola discutido no capítulo 2. A Figura 4.2, a seguir, apresentará o código necessário para isso.

Figura 4.2 – Gráfico do movimento da bola.



Fonte: Elaborada pelos autores.

Agora, vamos analisar cada célula.

Na célula (In[1:]), digitamos *using Plots*.

Na célula (In[2:]), atribuímos valores apenas para  $v_0 = 7$  e  $g$

= 9.81. Esse detalhe é muito importante, pois caso você atribua algum valor para  $t$ , como por exemplo 0.4, a saída do programa dirá que  $y$  já possui valor e conseqüentemente não plotará o gráfico. Isso porque não devemos atribuir um valor para  $t$ , mas um intervalo, como veremos mais adiante.

Observe que após a atribuição do valor de  $g$ , colocamos um “;”. Esse símbolo avisa a Julia que estamos atribuindo valores às variáveis, mas que não queremos que estes sejam exibidos na tela.

Já na célula (In[3]:), fizemos o seguinte em cada linha:

1. Inserimos a função que estamos analisando;
2. Digitamos **plot(y,0,0.4)**. Nessa linha de código, **y** é a nossa função e **0,0.4** é o intervalo de  $t$ . Esse comando é responsável por plotar o gráfico;
3. Na terceira e última linha, nomeamos os eixos **xlab = “t(s)”**, **ylab = “y(m)”**, e demos um título ao gráfico, **title= “Lançamento vertical”**. Perceba que utilizamos *plot!*. O símbolo **!** faz com que Julia execute os dois plots ao mesmo tempo. Caso não tivéssemos colocado este símbolo, Julia teria impresso na tela apenas o último **plot**.

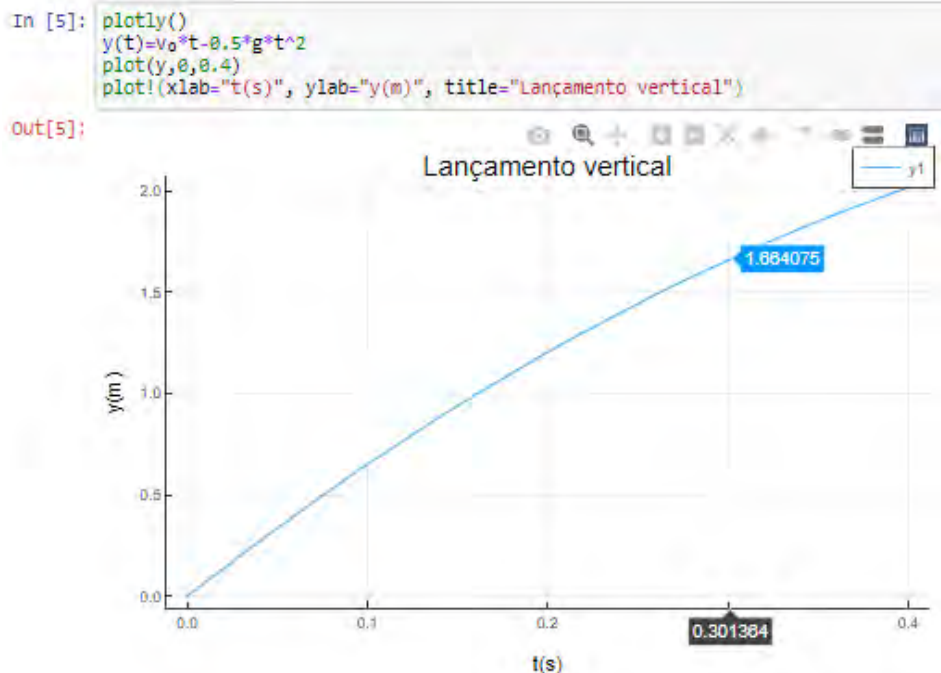
Agora, basta clicar em Run para que o gráfico seja gerado.

Se desejar, você poderá tornar seu gráfico interativo. Para isso, será necessário adicionar o *backend plotly()*. Os *backends*<sup>4</sup> dão “vida” ao Plots! O *plotly()*, por exemplo, permite que

4. <https://docs.juliaplots.org/latest/backends/>

informações dos pontos fiquem visíveis quando você passar o mouse sobre o gráfico. Na Figura 4.3 apresentaremos um exemplo.

Figura 4.3 – Gráfico interativo com o `plotly()`.



Fonte: Elaborada pelos autores.

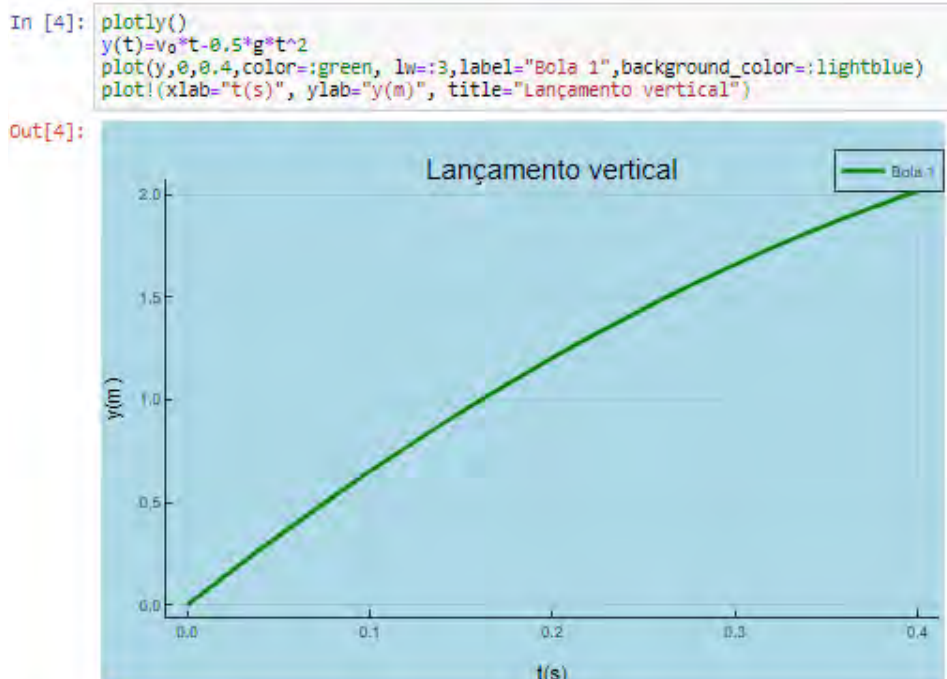
O que há de novo nesse gráfico em relação ao anterior?

Primeiro, como já havíamos digitado *using Plots* para gerar o gráfico anterior, nesse novo gráfico não houve a necessidade de digitá-lo novamente, já que o pacote Plots foi carregado. Outra diferença está na primeira linha de (In[5]:), onde incluímos o *backend* **plotly()**.

Observe, em (Out[5]:), que ao passarmos o mouse sobre o ponto (0.301364, 1.664075), ele ficou em destaque nas cores preto e azul, respectivamente. Outro detalhe na imagem são os ícones que aparecem na parte superior da figura. Ao clicarmos no primeiro ícone, que possui a imagem de uma câmera fotográfica, é feito o *download* da imagem. Percorrendo por cada um desses ícones, ainda é possível aplicar *zoom* na imagem e dar *reset* nas mudanças feitas. Vá em frente e teste-os!

Você ainda tem muitas opções, como por exemplo: modificar a cor da linha do gráfico, sua espessura ou a cor de fundo. Na Figura 4.4, apresentaremos um exemplo.

Figura 4.4 – Acrescentando opções ao gráfico.



Fonte: Elaborada pelos autores.

Veja, no primeiro plot, na terceira linha de (In[4]:), que acrescentamos a cor verde (color=:green) à linha do gráfico e modificamos sua espessura (line width) (lw=:3). Inserimos também uma legenda (label="Bola 1") e alteramos a cor de fundo (background\_color=:lightblue).

Antes de seguir para a próxima seção, sugiro que você customize o seu gráfico.

## 4.2 Resolvendo o problema do encontro de móveis com o Plots

A análise do encontro de móveis é uma situação bastante comum em Física básica. Vejamos um exemplo.

Dois carros, A e B, estão se movimentando sobre uma mesma trajetória retilínea, segundo as funções horárias  $S_A = 70 \cdot t$  e  $S_B = 200 - 30 \cdot t$ , com  $S$  em quilômetros e  $t$  em horas. Em que instante e posição da trajetória eles irão se encontrar?

Na Figura 4.5 apresentaremos o código e o gráfico deste exemplo.

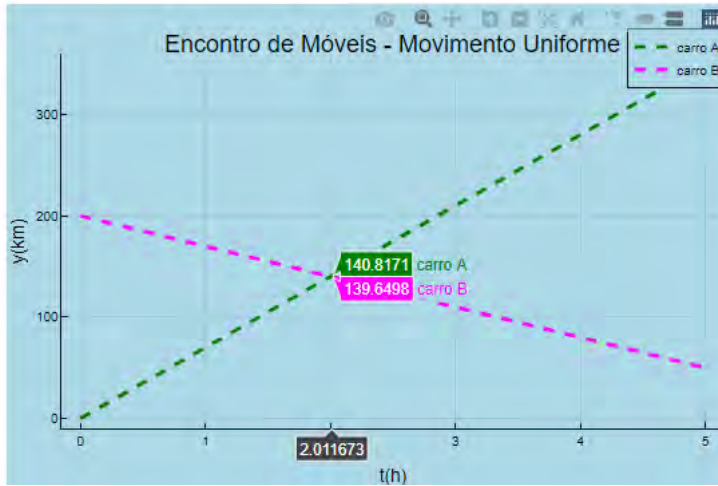
Figura 4.5 – Encontro de Móveis.

```
In [1]: using Plots

In [3]: plotly()
a(t) = 70*t
b(t) = 200-30*t
plot(a,0,5,label="carro A", color=:green, lw=3, background_color=:lightblue, linestyle=:dash)
plot!(b,0,5,label="carro B", color=:magenta, lw=3, linestyle=:dash)
plot!(xlabel="t(h)", ylabel="y(km)")
title!("Encontro de Móveis - Movimento Uniforme")
```



Out[3]:



Fonte: Elaborada pelos autores.

O que há de novo neste gráfico em relação ao anterior?

Além de termos duas funções horárias, uma para o carro A e outra para o carro B, incluímos a opção estilo de linha **linestyle** (`linestyle=:dash`). Existem outras opções para estilo de linha, como `:solid` ou `:dot`.

Outra mudança está na última linha, em que inserimos o comando **title!**. Esse comando solicita ao Plots que dê um título ao gráfico. No caso, o título que escolhemos foi “Encontro de Móveis - Movimento Uniforme”.

Com o gráfico gerado, você perceberá facilmente que, aproximadamente duas horas após o início do movimento, os carros A e B se encontrarão no quilômetro 140.

## 4.3 Arrays

Nesta seção, falaremos de duas estruturas: vetores e matrizes, também conhecidos como *Arrays*. A diferença entre eles é que o vetor é uma array unidimensional que pode ser escrito da forma  $M \times 1$ , enquanto a matriz é um array multidimensional que pode ser escrito na forma  $M \times N$ . Ambas armazenam dados de forma ordenada e entre colchetes [...]. Vejamos alguns exemplos de vetores e matrizes na Figura 4.6.

Figura 4.6 – Sintaxe de vetores e matrizes em Julia.

```
In [8]: a = [1,2,3,4,5] # Isso é um vetor.
```

```
Out[8]: 5-element Array{Int64,1}:  
 1  
 2  
 3  
 4  
 5
```

```
In [7]: mat1 = [1 2 3; 4 5 6] # Isso é uma matriz
```

```
Out[7]: 2x3 Array{Int64,2}:  
 1 2 3  
 4 5 6
```

Fonte: Elaborada pelos autores.

Perceba, em (In[8:]), que em um vetor, os elementos são separados por vírgula, enquanto que na matriz, apresentada em (In[7:]), as linhas são separadas por ponto e vírgula e as colunas, por espaços. Observe também que para o vetor *a*, o programa retornou automaticamente a informação **5-element**

**Array{Int64,1}**. Isso significa que o vetor possui 5 elementos do tipo inteiro e é um array de 1 (uma) dimensão.

**Cuidado!** Em programação um vetor é simplesmente uma lista de valores, não tendo a *priori* um significado geométrico, como sua definição matemática.

Por sua vez, observe que para a matriz, o programa retornou a informação **2x3 Array{Int64,2}**. Isso significa que a matriz é do tipo 2x3, com elementos do tipo inteiro e possui 2 dimensões. Um detalhe interessante é que as operações matriciais em Julia são bem parecidas com as notações matemáticas tradicionais. Vejamos os exemplos a seguir:

**Exemplo 1:**

Dadas as matrizes  $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  e  $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ , calcule  $A \cdot B$ .

A solução deste exemplo encontra-se na Figura 4.7.

Figura 4.7 - Resolução do exemplo 1 - Produto de Matrizes.

```
In [10]: #Produto de Matrizes
A=[1 2;3 4]
B=[5 6;7 8]
A*B
```

```
Out[10]: 2x2 Array{Int64,2}:
 19  22
 43  50
```

Fonte: Elaborada pelos autores.

Observe, em (In[10]:), que o produto da matriz  $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$  pela matriz  $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$  gerou a matriz  $A \cdot B = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$ , do tipo 2x2.

**Exemplo 2:**

Sendo  $A = \begin{bmatrix} 1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$ ,  $B = \begin{bmatrix} 2 & -1 \\ 1 & 0 \end{bmatrix}$  e  $A^t$ , a matriz transposta de  $A$ , determine o valor de  $A^t \cdot B$ .

A solução deste exemplo encontra-se na Figura 4.8.

Figura 4.8 – Resolução do exemplo 2 – Matrizes transpostas.

```
In [11]: #Calculo do produto da transposta da matriz A pela matriz B
A=[1 2 -1;0 -1 2]
B=[2 -1;1 0]
A'*B # A sintaxe da transposta de A é A'.
```

```
Out[11]: 3x2 Array{Int64,2}:
 2  -1
 3  -2
 0   1
```

Fonte: Elaborada pelos autores.

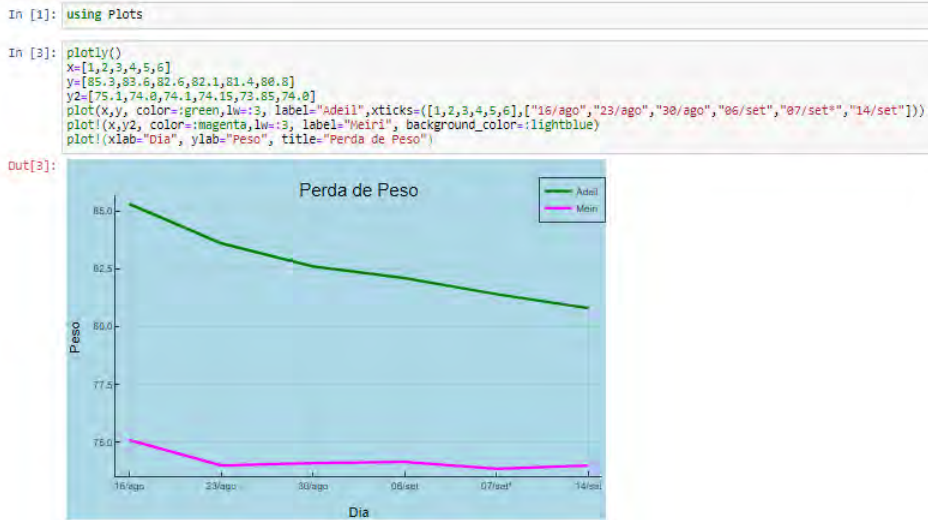
O produto da transposta de  $A$  pela matriz  $B$  gerou uma matriz do tipo  $3 \times 2$ .

Na próxima subseção, utilizaremos o conceito de vetor para gerarmos um gráfico com o pacote Plots.

### 4.3.1 Acompanhando a perda de peso

No decorrer do ano de 2020, durante a pandemia de COVID-19, ganhamos um peso extra. Resolvemos então, fazer um regime. Como forma de nos motivarmos, criamos o gráfico da nossa perda de peso, semana a semana. O resultado das quatro primeiras semanas é apresentado na Figura 4.9.

Figura 4.9 – Acompanhando a perda de peso.



Fonte: Elaborada pelos autores.

Agora, vamos interpretar linha a linha deste gráfico.

1. Em (In[1]:), carregamos o pacote Plots usando *using*;
2. Na primeira linha de (In[3]:), informamos a Julia que iremos utilizar o *backend* Plotly();
3. Nas três linhas seguintes, criamos os vetores (x, y, y2). O vetor x está armazenando os seis dias em que aconteceram as pesagens, já os vetores y e y2 estão armazenando nossos pesos;
4. Na quinta linha, temos uma novidade: trata-se do **xticks**. Ele é usado para que o código substitua os valores do vetor x pelas datas ["16/ago","23/ago","30/ago","06/set","07/set\*","14/set"];
5. Na sexta linha, solicitamos a criação de um novo gráfico indicando sua cor, espessura de linha e cor do plano de fundo;
6. Por fim, na última linha, nomeamos os eixos x e y e demos um título ao gráfico.





### Importando e exportando dados

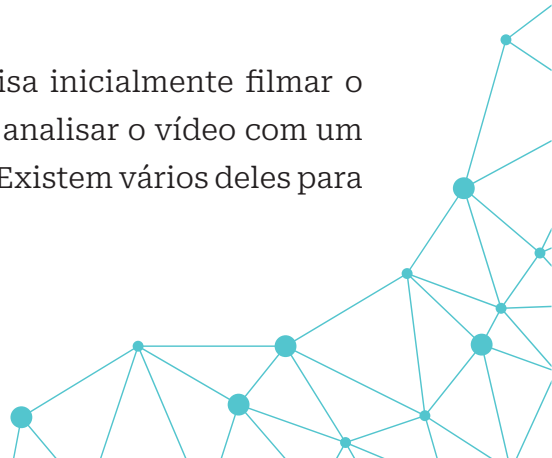
Neste capítulo, falaremos um pouco sobre como importar e exportar dados para Julia utilizando dois pacotes: CSV<sup>1</sup> e DataFrames.<sup>2</sup> Vejamos a situação a seguir.

Suponha que foi dada a você a tarefa de analisar o movimento de uma bola de tênis, arremessada obliquamente no ar, e de entregar um relatório sobre o estudo desse movimento. Seria interessante que em seu relatório constasse não apenas a revisão teórica, mas também tabelas e gráficos obtidos a partir da análise do movimento real desse objeto.

Para obter dados reais, você precisa inicialmente filmar o movimento do objeto e posteriormente analisar o vídeo com um *software* ou aplicativo de rastreamento. Existem vários deles para

1. <http://juliadata.github.io/CSV.jl/v0.1.1/>

2. <https://juliapackages.com/p/dataframes>



os mais diversos sistemas, aqui vamos listar três bastante úteis. O primeiro é o aplicativo *VidAnalysis*<sup>3</sup> para dispositivo Android, o segundo é o aplicativo *Vernier Video Physics*<sup>4</sup>, para iPhone e iPad e o terceiro é o *software Tracker*<sup>5</sup>, que está disponível para Windows, Mac e Linux. Esses aplicativos e *software* geram, automaticamente, gráficos, tabelas e um arquivo CSV<sup>6</sup>, após o processo de análise do vídeo<sup>7</sup>. Para uma introdução rápida à videoanálise, dirija-se ao Apêndice B.

Caso queira personalizar seus gráficos, você deve enviar o arquivo CSV para o seu e-mail e, posteriormente, baixar no computador. Em seguida, basta fazer o *upload* para o Jupyter *Notebook*. Na próxima seção, veremos como isso acontece na prática.

## 5.1 Pacote CSV

Para importar um arquivo CSV para o Jupyter Notebook, basta clicar em *upload*, no canto superior direito do painel de controle (*dashboard*)<sup>8</sup> e em seguida selecionar o arquivo. No nosso caso, o arquivo é o *bola.csv*<sup>9</sup>, como pode ser visto na Figura 5.1.

3. <https://play.google.com/store/apps/details?id=com.vidanalysis>

4. <https://apps.apple.com/br/app/vernier-video-physics/id389784247>

5. <https://www.physlets.org/tracker/>

6. Formato de arquivo que armazena dados tabelados.

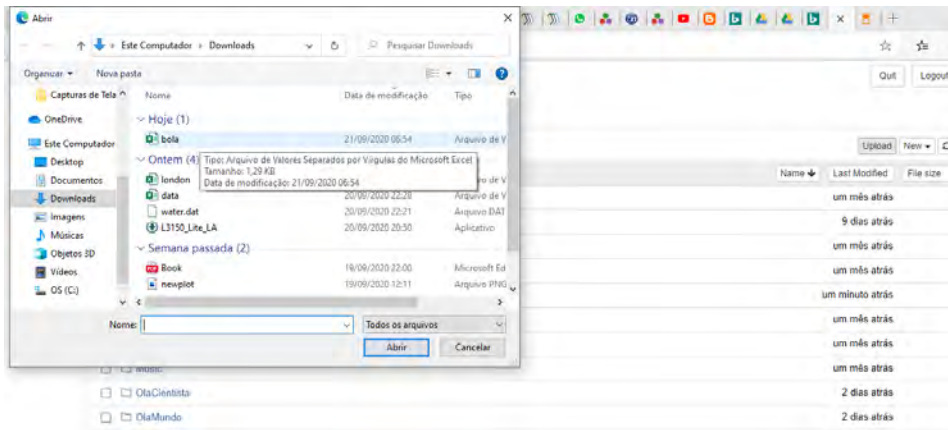
7. O vídeo utilizado nos exemplos deste capítulo está disponível para download em: <https://bit.ly/33Vu9c6>

8. Caso esteja utilizando o Jupyter online você precisará clicar no ícone do Jupyter, localizado no canto superior esquerdo da tela, para ter acesso ao *dashboard*.

9. sse arquivo encontra-se disponível para download em: <https://bit.ly/34hrMjP>



Figura 5.1 – Upload do arquivo bola.csv para o Jupyter Notebook.



Fonte: Elaborada pelos autores.

Após o *upload*, o arquivo ficará disponível no próprio painel de controle.

Para utilizar os dados armazenados no arquivo, primeiro você precisará instalar o pacote CSV seguindo as instruções apresentadas no capítulo 3. Em seguida, você deverá carregar o pacote e importar o arquivo para seu código, como mostra a Figura 5.2.

Figura 5.2 – Usando o pacote CSV e importando o arquivo bola.csv.

```
In [2]: using CSV
```

```
In [3]: bola=CSV.read("bola.csv", normalizenames=true)
```

```
Out[3]: 27 rows × 5 columns
```

	time_s	x_distance_m	y_distance_m	Øx_velocity_m_s	Øy_velocity_m_s
	Float64	Float64	Float64	Float64?	Float64?
1	0.0	0.0854	0.1743	missing	missing
2	0.033	0.1957	0.2099	3.3423	1.0782
3	0.067	0.2917	0.3273	2.8254	3.4533
4	0.1	0.4092	0.4198	3.5579	2.8032

5	0.133	0.491	0.491	2.4798	2.1563
6	0.167	0.5942	0.5444	3.0347	1.5697
7	0.2	0.7045	0.612	3.3423	2.0485
8	0.234	0.8254	0.644	3.5579	0.9418
9	0.267	0.9535	0.6831	3.8814	1.186
10	0.3	0.9678	0.676	0.4313	-0.2156
11	0.334	1.1706	0.6547	5.9647	-0.6279
12	0.367	1.2986	0.6404	3.8814	-0.4313

Fonte: Elaborada pelos autores.

Agora, vamos explicar os códigos inseridos em cada célula:

1. Em (In[2]:), carregamos o pacote para nosso código utilizando *using CSV*;
2. Em (In[3]:), importamos o arquivo `bola.csv` usando a função **CSV.read**. Essa função lê os dados do arquivo e os armazena na variável `bola`;
3. Após clicar em **Run**, Julia retornou em (Out[3]:) uma tabela com todos os dados do movimento da bola.

Os nomes das colunas (`time [s]`, `y-distance [m]`), ocasionalmente, contém espaços e símbolos que são considerados caracteres inválidos. Para corrigir esse problema, incluímos no código o trecho “`normalizenames = true`”, que substituirá os caracteres inválidos por sublinhados, garantindo assim, que cada coluna seja um identificador válido em Julia. Os nomes das colunas podem ser vistos em (Out[3]:).

Na próxima seção, construiremos gráficos utilizando dados dessa tabela.

## 5.2 Construindo gráficos com dados de um arquivo CSV

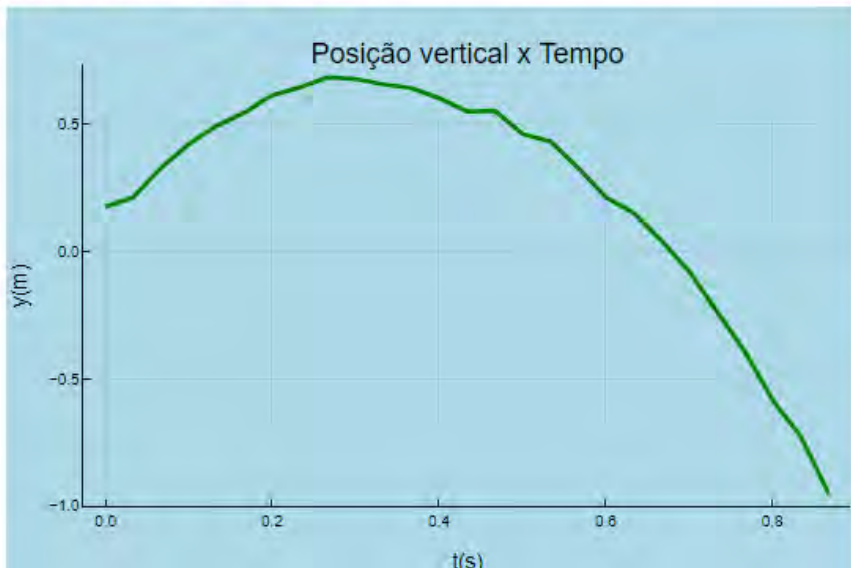
Utilizando a tabela importada na seção anterior, construiremos inicialmente o gráfico da posição vertical da bola em função do tempo. Para isso, carregamos o pacote Plots e escrevemos o código que gerará o gráfico, como mostra a Figura 5.3.

Figura 5.3 – Construção do gráfico da posição vertical em função do tempo.

```
In [3]: using Plots
```

```
In [5]: plotly()
plot(bola.time_s_, bola.y_distance_m,color=:green,lw=:3, label=false)
plot!(xlab="t(s)", ylab="y(m)", background_color=:lightblue)
title!("Posição vertical x Tempo")
```

```
Out[5]:
```



Fonte: Elaborada pelos autores.

Agora vamos explicar o que está acontecendo:

1. Em (In[3:]), carregamos o pacote Plots usando *using Plots*;

2. Na primeira linha em (In[5]:), instalamos o *backend Plotly()* para tornar nosso gráfico interativo;

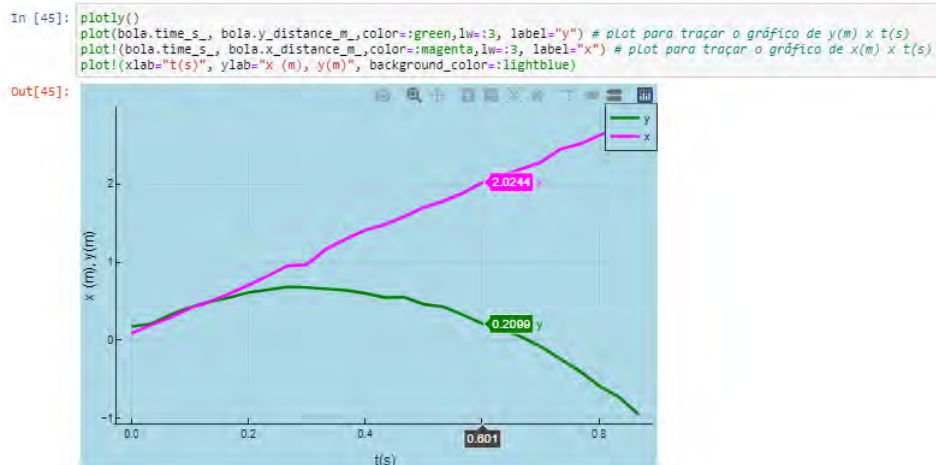
3. Na linha seguinte, incluímos o trecho do código responsável por gerar o gráfico da posição vertical em função do tempo. Esse trecho contém a principal diferença entre esse gráfico e os outros que já construímos. Nele, indicamos os dados para os eixos x e y, a cor, a espessura da linha, além de indicarmos que o gráfico não deverá conter legenda;

4. Na terceira linha, nomeamos os eixos x e y e mudamos a cor de fundo do gráfico;

5. Por fim, nomeamos o gráfico por meio do comando **title!**.

Caso queira incluir o gráfico da posição horizontal em função do tempo, basta inserir uma nova linha com os dados do gráfico e modificar os títulos dos eixos. Essas alterações encontram-se nas linhas 3 e 4 da célula (In[45]:) apresentada na Figura 5.4.

Figura 5.4 – Gráfico da posição (x,y) pelo tempo para o lançamento oblíquo da bola de tênis.



Fonte: Elaborada pelos autores.

Note que no movimento da bola há duas componentes cartesianas: uma na direção horizontal (x) e outra na direção vertical (y). No eixo horizontal, a aceleração é nula, enquanto que no eixo vertical, a aceleração, devido à gravidade, atua para baixo no sentido negativo.

Esses dois movimentos são independentes, o que significa dizer que um não afeta o outro. Além disso, observe que o primeiro, representado pela cor magenta, descreve um movimento uniforme, enquanto que o segundo, um movimento uniformemente variado.

Caso queira uma imagem com vários gráficos do movimento, cada um com seu plano cartesiano, basta seguir as orientações apresentadas na Figura 5.5.

Figura 5.5 – Gráficos do movimento da bola de tênis.

```
In [7]: plotly()
p1=plot(bola.time_s_, bola.x_distance_m_, color=:green, lw=:3, xlabel="t(s)", ylab="x(m)",
        title="Posição horizontal x Tempo",label=false)

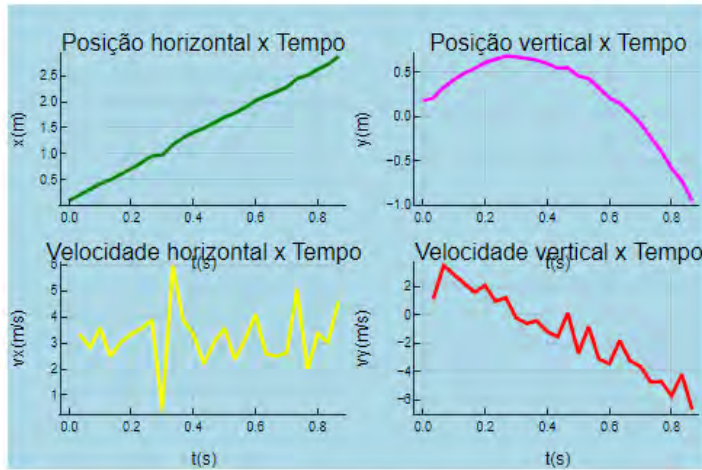
p2=plot(bola.time_s_, bola.y_distance_m_, color=:magenta, lw=:3, xlabel="t(s)", ylab="y(m)",
        title="Posição vertical x Tempo", label=false)

p3=plot(bola.time_s_, bola.Øx_velocity_m_s_, color=:yellow, lw=:3, xlabel="t(s)", ylab="vx(m/s)",
        title="Velocidade horizontal x Tempo", label=false)

p4=plot(bola.time_s_, bola.Øy_velocity_m_s_, color=:red,lw=:3, xlabel="t(s)", ylab="vy(m/s)",
        title="Velocidade vertical x Tempo", label=false)

plot(p1,p2,p3,p4, background_color=:lightblue, layout = (2, 2), legend = false)
```

out[7]:



Fonte: Elaborada pelos autores.

Observe que a diferença desse código para os anteriores é que nomeamos cada plot com p1, p2, p3 e p4 e na última linha, os agrupamos com uma especificação de *layout*.

### 5.3 Pacote DataFrames

Nesta seção, descreveremos mais um exemplo de como importar e exportar dados em Julia, agora incluindo o pacote DataFrames<sup>10</sup>.

DataFrames são estruturas que representam tabelas de dados, sendo que cada coluna dessa tabela é um vetor. Nosso objetivo agora é criar um DataFrame e exportá-lo para um arquivo CSV. Vejamos um exemplo.

Suponha que seu professor de Física pediu que fosse feito um levantamento histórico das descobertas das partículas

10. [http://juliadata.github.io/DataFrames.jl/v0.11/man/getting\\_started.htm](http://juliadata.github.io/DataFrames.jl/v0.11/man/getting_started.htm)

atômicas e subatômicas. O primeiro passo para realizar essa tarefa é preparar um conjunto de dados contendo, por exemplo, as seguintes variáveis: nome da partícula, ano que foi descoberta e o cientista que a descobriu. Cada uma dessas variáveis estará em colunas distintas. De posse desses dados, vejamos na Figura 5.6 o que você precisará para gerar um arquivo CSV.

Figura 5.6 – Gerando um arquivo CSV a partir de um DataFrame.

```
In [1]: using DataFrames
using CSV

In [2]: df = DataFrame(Particula = ["Raio-X", "Eletron", "Particula alfa", "Raio gama", "Nucleo atómico"],
                    Ano = [1895, 1897, 1899, 1900, 1911],
                    Cientista = ["Wilhelm Rontgen", "JJ Thomson", "Ernest Rutherford", "Paul Villard", "Ernest Rutherford"]
                    )
print(df)

5x3 DataFrame
┌───┬──────────┬───┬──────────┐
│ Row │ Particula │ Ano │ Cientista │
│   │ String    │ Int64 │ String    │
├───┼──────────┼───┼──────────┤
│ 1   │ Raio-X    │ 1895 │ Wilhelm Rontgen │
│ 2   │ Eletron   │ 1897 │ JJ Thomson      │
│ 3   │ Particula alfa │ 1899 │ Ernest Rutherford │
│ 4   │ Raio gama │ 1900 │ Paul Villard    │
│ 5   │ Nucleo atómico │ 1911 │ Ernest Rutherford │
└───┴──────────┴───┴──────────┘

In [3]: CSV.write("C:\\Users\\adeil\\Downloads\\particulas.csv", df)
Out[3]: "C:\\Users\\adeil\\Downloads\\particulas.csv"
```

Fonte: Elaborada pelos autores.

Vamos entender cada linha do código:

1. Em (In[1]:), carregamos os pacote DataFrames<sup>11</sup> e CSV;
2. Em (In[2]:), criamos a variável **df** e a atribuímos um DataFrame. Esse DataFrame possui três vetores, que representarão as colunas de nossa tabela;
3. Em seguida, solicitamos ao programa imprimir a variável **df**. Observe que ela possui 5 linhas e 3 colunas;
4. Em (In[3]:), utilizamos a função **CSV.write** para indicar o local onde o arquivo será salvo, além de solicitarmos que o DataFrame seja exportado para um arquivo CSV. Observe que

11. Antes de carregar o pacote DataFrames, você deverá instalá-lo seguindo as orientações do capítulo 3.

a função **CSV.write** possui dois parâmetros: o **C**, que indicará o caminho para a pasta onde o arquivo será salvo e **df**, que representará o DataFrame a ser salvo. O nome **particulas** é o novo arquivo a ser criado e **.csv** é a extensão de arquivo.<sup>12</sup>

É importante destacar que usamos barras duplas invertidas em **C** para evitar erros. Outro ponto a que chamamos sua atenção é que, caso você esteja utilizando o Jupyter *online*, o arquivo csv não será baixado automaticamente em seu computador. Para realizar o *download*, você deverá ir até o *dashboard*, clicar no nome do local onde o arquivo será salvo, escolher o file e, em seguida, clicar em *download*.

12. Para abrir esse arquivo, indicamos o Google Planilhas.





Funções, em programação, são bem parecidas com as funções matemáticas. Ou seja, ao defini-las, devemos nomeá-las, e se necessário, indicar os argumentos para que uma tarefa seja executada. Explicaremos argumentos e tarefas nos exemplos deste capítulo.

Existem três maneiras de se declarar funções em Julia. A primeira, por meio da estrutura “*function-end*”, em que se inicia o código com o nome *function* e se finaliza com a palavra-chave *end*, como mostra a Figura 6.1.

Figura 6.1 – Definição de função.

```
In [4]: function soma(arg1,arg2)
        return arg1 + arg2
        end

Out[4]: soma (generic function with 1 method)

In [5]: soma(3,2)

Out[5]: 5
```

Fonte: Elaborada pelos autores.



Observe, na figura, que declaramos a função **soma** seguida de dois argumentos: o **arg1** e o **arg2**. Essa função retornará ao usuário o valor da expressão avaliada, no caso, **arg1 + arg2**. Em (In[5]:), atribuímos aos argumentos os valores 3 e 2. A função, então, nos retornou, em (Out[5]:), a soma deles.

Chamamos sua atenção para a segunda linha de (In[4]:). Essa linha representa o corpo da função e deverá ser recuada. Esse recuo é chamado de endentação. Em Julia, o recuo deve ser de 4 espaços ou 1 [TAB], porém essas regras não são tão rigorosas. Recomenda-se, no entanto, que você não misture [TAB] e espaços enquanto recua o código.

A segunda maneira de declarar uma função é muita parecida com a estrutura de uma função na matemática, ou seja, escolhemos um nome para ela, passamos os argumentos e atribuímos a tarefa que ela deverá realizar, utilizando o símbolo “=”. Você entenderá melhor o que estamos dizendo, observando a Figura 6.2

Figura 6.2 – Definindo uma função em apenas uma linha.

```
In [19]: soma2(arg1,arg2)=arg1+arg2
Out[19]: soma2 (generic function with 1 method)

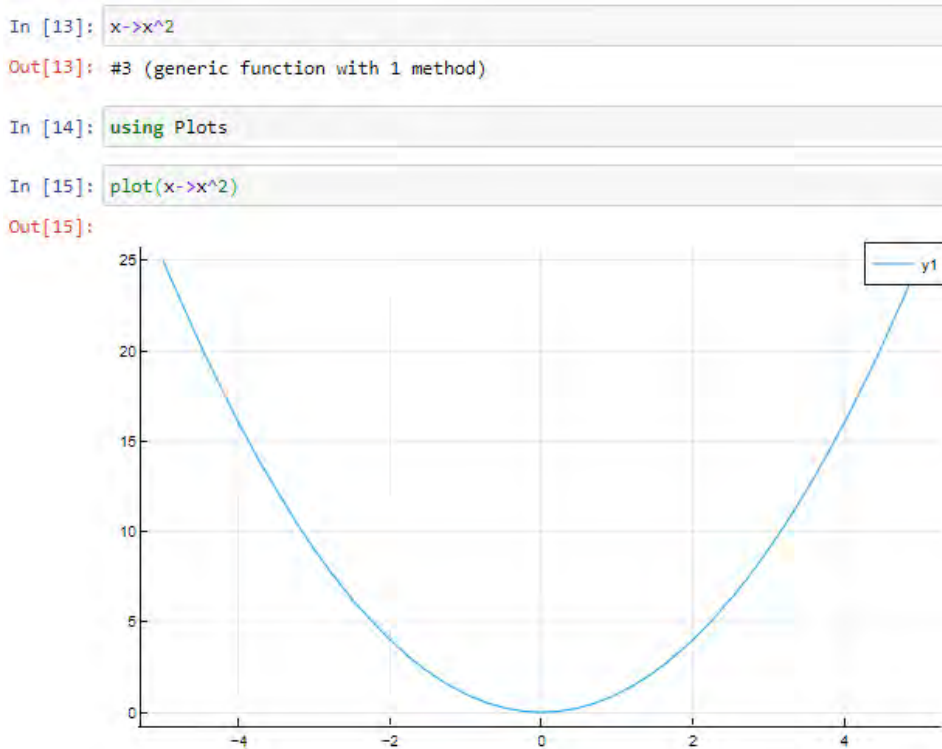
In [20]: soma2(3,2)
Out[20]: 5
```

Fonte: Elaborada pelos autores.

Perceba que declaramos a função em uma única linha e o retorno foi o mesmo. Em alguns casos, é mais conveniente declarar funções dessa forma.

Por fim, podemos criar funções anônimas. Essas funções são chamadas assim porque não possuem um nome definido em sua declaração. Veja sua sintaxe na Figura 6.3.

Figura 6.3 – Definindo uma função anônima.



Fonte: Elaborada pelos autores.

Como você pode ver em (In[13]:), os argumentos da função são seguidos do operador `->` e da tarefa que ela deverá realizar. Geralmente, as funções anônimas são passadas como argumentos para outras funções, como pode ser visto em (In[15]:), em que a passamos como argumento para a função `plot`. O retorno é o gráfico de  $x^2$  como se verifica em (Out[15]:).

## 6.1 Calculando o peso e inserindo as unidades de medidas

Sabemos que o peso de um corpo é igual ao produto da sua massa pela aceleração gravitacional. No entanto, dependendo do local onde o corpo está localizado, a aceleração gravitacional pode variar. Quanto mais o corpo se afasta da Terra, menor ela será. Dessa forma, caso você esteja no telhado de sua casa, você pesará um pouquinho menos.

Nesta seção criamos uma função que calcula o peso de um corpo. Ela possui dois argumentos: a massa e a gravidade. Veja como definimos nossa função e alguns exemplos na Figura 6.4.

Figura 6.4 – Função que calcula o peso.

```
In [1]: function peso(m,g=9.81)
        return m*g
        end

Out[1]: peso (generic function with 2 methods)

In [2]: peso(80,9.81) # Peso na Terra (g=9.81 m/s^2.)

Out[2]: 784.8000000000001

In [3]: peso(80,3.72) # Peso em Marte (g=3.72 m/s^2.)

Out[3]: 297.6

In [4]: peso(80,1.6) # Peso na Lua (g=1.6 m/s^2.)

Out[4]: 128.0
```

Fonte: Elaborada pelos autores.

Em nosso exemplo calculamos o peso em três lugares diferentes: na Terra, em Marte e na Lua. Repare que em nenhuma das saídas (Out[ ]) estão indicadas as unidades de medidas. Podemos resolver esse problema utilizando o pacote Unitful, estudado no capítulo 3. Veja o resultado na Figura 6.5.

Figura 6.5 – Função que calcula o peso agregada ao pacote Unitful.

```
In [5]: using Unitful

In [6]: peso(80u"kg",9.81u"m/s^2") # Peso na Terra (g=9.81 m/s^2.)
Out[6]: 784.8000000000001 kg m s^-2

In [7]: peso(80u"kg",3.72u"m/s^2") # Peso em Marte (g=3.72 m/s^2.)
Out[7]: 297.6 kg m s^-2

In [8]: peso(80u"kg",1.6u"m/s^2") # Peso na Lua (g=1.6 m/s^2.)
Out[8]: 128.0 kg m s^-2
```

Fonte: Elaborada pelos autores.

Após carregarmos o pacote Unitful, em (In[5]:), repetimos os exemplos anteriores. Contudo, desta vez, inserimos as unidades de medidas para a massa e aceleração gravitacional para que o programa nos retornasse os resultados em  $\text{kg}\cdot\text{m}/\text{s}^2$ , que é equivalente a newtons (N).

## 6.2 Calculando o Trabalho de uma força

Sabemos que o trabalho ( $W$ ) é a medida da energia que uma força transfere em um deslocamento. Ele é calculado por meio da seguinte expressão:

$$W = F \cdot d \cdot \cos(\theta) \quad (6.1)$$

Onde  $F$  representa a força;  $d$ , o deslocamento e  $\theta$ , o ângulo entre a força e o deslocamento.

Como os ângulos em Julia são calculados em radianos e precisamos do valor do cosseno em graus, podemos seguir dois caminhos distintos para realizar essa conversão:

1. Multiplicar o valor em radianos por  $\pi/180$ ;
2. Inserir a função trigonométrica que se deseja utilizar seguida pela letra d (degree).

De posse dessas informações, vamos analisar o exemplo apresentado na Figura 6.6.

Figura 6.6 – Função que calcula o trabalho de uma força usando `cosd`.

```
In [59]: function calcule_trabalho(F, d, θ) #para digitar a letra grega teta faça \theta[TAB]
          return F*d*cosd(θ)
        end

Out[59]: calcule_trabalho (generic function with 1 method)

In [60]: calcule_trabalho(10, 5, 30)

Out[60]: 43.30127018922193
```

Fonte: Elaborada pelos autores.

Utilizando a estrutura “*function-end*”, criamos a função **calcule\_trabalho**. Veja, na célula (In[59]:), que ela possui três argumentos: força, deslocamento e ângulo  $\theta$ . Observe, também, que escolhemos o caminho 2, inserindo a letra **d** na função cosseno. Já na célula (In[60]:), inserimos valores para cada um desses argumentos.

*Caso tivéssemos escolhido usar a função `cos`, teríamos que utilizar o caminho 1, indicando como terceiro parâmetro o valor de  $30.\pi/180$  como mostra a Figura 6.7.*

Figura 6.7 – Função que calcula o trabalho de uma força usando  $\cos$ .

```
In [7]: function calcule_trabalho(F, d,  $\theta$ ) #para digitar a letra grega teta faça \theta[TAB]
        return F*d*cos( $\theta$ )
        end

Out[7]: calcule_trabalho (generic function with 1 method)

In [8]: calcule_trabalho(10, 5, 30*\pi/180) #Para escrever  $\pi$  você deve digitar \pi[TAB]

Out[8]: 43.30127018922194
```

Fonte: Elaborada pelos autores.

Caso queira incluir as unidades de medidas em seu resultado, basta seguir os passos apresentados na seção 6.1. A Figura 6.8 apresenta os códigos necessários para isso.

Figura 6.8 – Função que calcula o trabalho de uma força integrada ao pacote Unitful.

```
In [1]: using Unitful

In [2]: function calcule_trabalho(F, d,  $\theta$ ) #para digitar a letra grega teta faça \theta[TAB]
        return F*d*cos( $\theta$ )
        end

Out[2]: calcule_trabalho (generic function with 1 method)

In [3]: calcule_trabalho(10u"N", 5u"m", 30*\pi/180) #Para escrever  $\pi$  você deve digitar \pi[TAB]

Out[3]: 43.30127018922194 m N
```

Fonte: Elaborada pelos autores.

Como você pode ver, em (Out[3]:), nosso resultado apresenta unidades em  $\text{N} \cdot \text{m}$ .

## 6.3 Argumentos

O argumento em uma função pode ser do tipo obrigatório<sup>1</sup> ou opcional. Vamos discutir esses conceitos observando a Figura 6.9.

1. Um detalhe importante é que os argumentos obrigatórios devem aparecer antes dos opcionais.



Figura 6.9 – Função que calcula o peso com  $g$  como argumento opcional.

```
In [1]: function peso(m,g=9.81)
        return m*g
        end

Out[1]: peso (generic function with 2 methods)

In [2]: peso(70)

Out[2]: 686.7

In [3]: peso(70,3.72) # A gravidade em Marte é 3.72 m/s^2.

Out[3]: 260.40000000000003
```

Fonte: Elaborada pelos autores.

Um argumento é chamado de obrigatório quando você é obrigado a passar o seu valor. Em (In[1]:), perceba que o valor da massa ( $m$ ) é desconhecido, logo, você precisará indicar seu valor para que a função seja executada.

Por sua vez, um argumento é chamado de opcional quando ele possui algum valor conhecido, como por exemplo  $g$ , que possui o valor igual a 9.81. Em (In[2]:), quando chamamos a função `peso`, apenas o valor de 70 foi repassado como argumento. Esse valor representará a massa, já que conhecemos o valor de  $g$ . Caso você tivesse indicado um valor também para  $g$ , este iria sobrepor o valor de 9.81. Veja um exemplo na célula (In[3]:).

Às vezes é necessário, para dar uma maior clareza ao código, chamar o argumento pelo seu nome. Quando isso acontece chamamos esses argumentos de palavras-chave. Importante

destacar que eles devem aparecer por último na definição da função e separados por ponto e vírgula (;). Para exemplificar, definimos a função altura e passamos os argumentos  $v_0$ ,  $g$  e  $t$ . Sendo  $v_0$  um argumento obrigatório e  $g$  e  $t$ , argumentos de palavras-chave, como mostra a Figura 6.10.

Figura 6.10 – Definido uma função com argumentos de palavra-chave.

```
In [78]: # Função para calcular a altura de uma bola em movimento vertical
function altura(v0;g=9.81,t=0.6)
    return v0*t - 0.5*g*t^2
end
```

```
Out[78]: altura (generic function with 1 method)
```

```
In [79]: altura(10)
```

```
Out[79]: 4.2341999999999995
```

```
In [81]: altura(10,g=3.72)
```

```
Out[81]: 5.3304
```

```
In [84]: altura(10,t=0.4)
```

```
Out[84]: 3.2152
```

```
In [82]: altura(10,g=1.6,t=10)
```

```
Out[82]: 20.0
```

---

Fonte: Elaborada pelos autores.

Vamos entender nosso código.

1. Na primeira linha, em (In[78]:), fizemos apenas um comentário;

2. Na segunda linha, definimos a função altura por meio da estrutura *function-end* e passamos seus argumentos. Repare que os dois últimos argumentos possuem valores padrão e estão

separados do primeiro por ponto e vírgula. Então, eles serão chamados de argumentos de palavra-chave;

3. Na terceira linha, solicitamos ao programa que nos retorne o cálculo  $y = v_0 \cdot t - 0,5 \cdot g \cdot t^2$ ;

4. Por fim, finalizamos a função com a palavra-chave **end**.

Definida nossa função, nas células seguintes apresentamos alguns exemplos.

No primeiro exemplo, em (In[79]:), chamamos a função altura e entre parênteses colocamos o valor 10. O programa entenderá automaticamente que o valor 10 refere-se ao argumento  $v_0$ , pois  $g$  e  $t$  possuem valores padrão, no caso  $g = 9.81$  e  $t = 0.6$ .

Já no segundo exemplo, em (In[81]:), passamos dois argumentos. Repare que o valor de  $v_0$  é 10 e o valor de  $g$  foi modificado para 3.72. Neste segundo exemplo, o programa entenderá que queremos atribuir um valor a  $v_0$ , substituir o valor de  $g$  e manter o *valor padrão* de  $t$ .

No terceiro exemplo, em (In[84]:), atribuímos à  $v_0$  o valor 10 e modificamos o valor de  $t$ , chamando-o explicitamente. Assim, o programa entenderá que queremos atribuir um valor a  $v_0$ , manter o *valor padrão* de  $g$  e substituir o valor de  $t$  por 0.4.

No quarto e último exemplo, em (In[82]:), passamos três argumentos. Repare que o valor de  $v_0$  é 10, o valor de  $g$  foi modificado para 1.6 e o valor de  $t$  modificado para 10. Aqui, o

programa entenderá que queremos atribuir um valor a  $v_0$  e substituir os valores de  $g$  e  $t$ .





Neste capítulo, discutiremos um dos conceitos mais importantes em Julia, que é o de *broadcasting* (transmissão). Para entendê-lo, voltaremos a falar sobre *arrays* (matrizes e vetores), assunto discutido no capítulo 4. Naquela ocasião, definimos a matriz A como sendo  $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ . Agora, vamos substituir essa matriz na função  $f(x) = x^2$ . Vejamos a Figura 7.1.

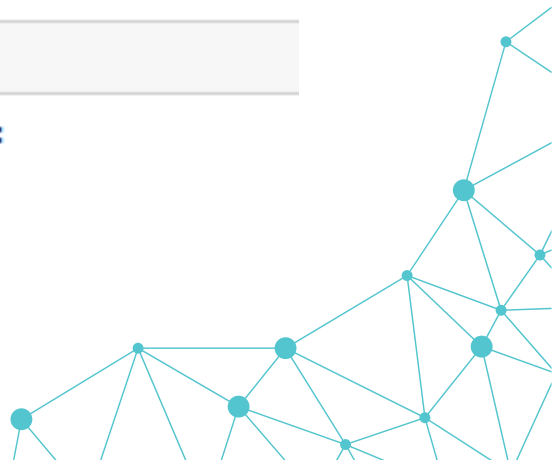
Figura 7.1 – Substituindo a matriz A na função  $f(x) = x^2$ .

```
In [33]: f(x)=x^2
```

```
Out[33]: f (generic function with 1 method)
```

```
In [34]: A=[1 2;3 4]
```

```
Out[34]: 2x2 Array{Int64,2}:  
 1  2  
 3  4
```



```
In [35]: f(A)
```

```
Out[35]: 2x2 Array{Int64,2}:  
 7  10  
15  22
```

```
In [36]: A*A
```

```
Out[36]: 2x2 Array{Int64,2}:  
 7  10  
15  22
```

Fonte: Elaborada pelos autores.

Vamos te explicar o que está acontecendo:

1. Inicialmente, definimos nossa função  $f(x) = x^2$ ;
2. Na segunda entrada, (In[34]:), definimos a matriz A;
3. Na terceira entrada, (In[35]:), substituímos a matriz A na função f. O resultado dessa operação é apresentado em (Out[35:]);
4. Inserimos em (In[36:]), o produto da matriz A por ela mesma para comprovar o resultado apresentado em (Out[35:]) e o resultado foi o mesmo.

Caso não queira o produto da matriz por ela mesma, mas a aplicação da função a cada elemento da matriz A, você deve utilizar o conceito de *broadcasting*, ou seja, você deve dizer ao código que quer que a função seja transmitida a cada elemento da matriz, como mostra a Figura 7.2.

Figura 7.2 – Aplicando o conceito de broadcasting a uma matriz.

```
In [33]: f(x)=x^2
Out[33]: f (generic function with 1 method)

In [34]: A=[1 2;3 4]
Out[34]: 2x2 Array{Int64,2}:
          1  2
          3  4

In [37]: f.(A)
Out[37]: 2x2 Array{Int64,2}:
          1  4
          9 16
```

Fonte: Elaborada pelos autores.

Perceba, em (In[37]:), que para utilizarmos o conceito de broadcasting, inserimos um ponto entre a função  $f$  e seu argumento.

Agora, definiremos o vetor  $a = [1, 2, 3, 4, 5]$  e tentaremos substituí-lo na função  $f(x) = x^2$  sem utilizar o conceito de *broadcasting*, como mostra a Figura 7.3.

Figura 7.3 – Aplicando o conceito de broadcasting a um vetor.

```
In [38]: a=[1,2,3,4,5]
Out[38]: 5-element Array{Int64,1}:
 1
 2
 3
 4
 5

In [39]: f(a)

MethodError: no method matching ^(::Array{Int64,1}, ::Int64)
Closest candidates are:
  ^(!Matched::BigFloat, ::Integer) at mpfr.jl:599
  ^(!Matched::Float64, ::Integer) at math.jl:899
  ^(!Matched::Float32, ::Integer) at math.jl:907
  ...

Stacktrace:
 [1] macro expansion at .\none:0 [inlined]
 [2] literal_pow at .\none:0 [inlined]
 [3] f(::Array{Int64,1}) at .\In[33]:1
 [4] top-level scope at In[39]:1
 [5] include_string(::Function, ::Module, ::String, ::String) at .\loading.jl:1091

In [40]: f.(a)
Out[40]: 5-element Array{Int64,1}:
 1
 4
 9
16
25
```

Fonte: Elaborada pelos autores.

Perceba, em (In[39]:), que o programa nos retornou um erro. Isso porque o quadrado de um vetor não é bem definido. No entanto, aplicando o conceito de *broadcasting* em (In[40]:), a função foi transmitida a cada elemento do vetor, ou seja, cada elemento do vetor foi elevado ao quadrado.

Tente outros valores!



## 7.1 Broadcasting em funções trigonométricas

Nesta seção, apresentaremos mais um exemplo da aplicação do conceito de *broadcasting*, mas agora utilizando a função trigonométrica seno. Vejamos a Figura 7.4.

Figura 7.4 – Aplicando o conceito de broadcasting à função trigonométrica Seno.

```
In [41]: a=[1,2,3,4,5]
Out[41]: 5-element Array{Int64,1}:
 1
 2
 3
 4
 5

In [42]: sin(a)
MethodError: no method matching sin(::Array{Int64,1})
Closest candidates are:
  sin(!Matched::BigFloat) at mpfr.jl:727
  sin(!Matched::Missing) at math.jl:1197
  sin(!Matched::Complex{Float16}) at math.jl:1145
  ...

Stacktrace:
 [1] top-level scope at In[42]:1
 [2] include_string(::Function, ::Module, ::String, ::String) at .\loading.jl:1091

In [43]: sin.(a)
Out[43]: 5-element Array{Float64,1}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
 -0.7568024953079282
 -0.9589242746631385
```

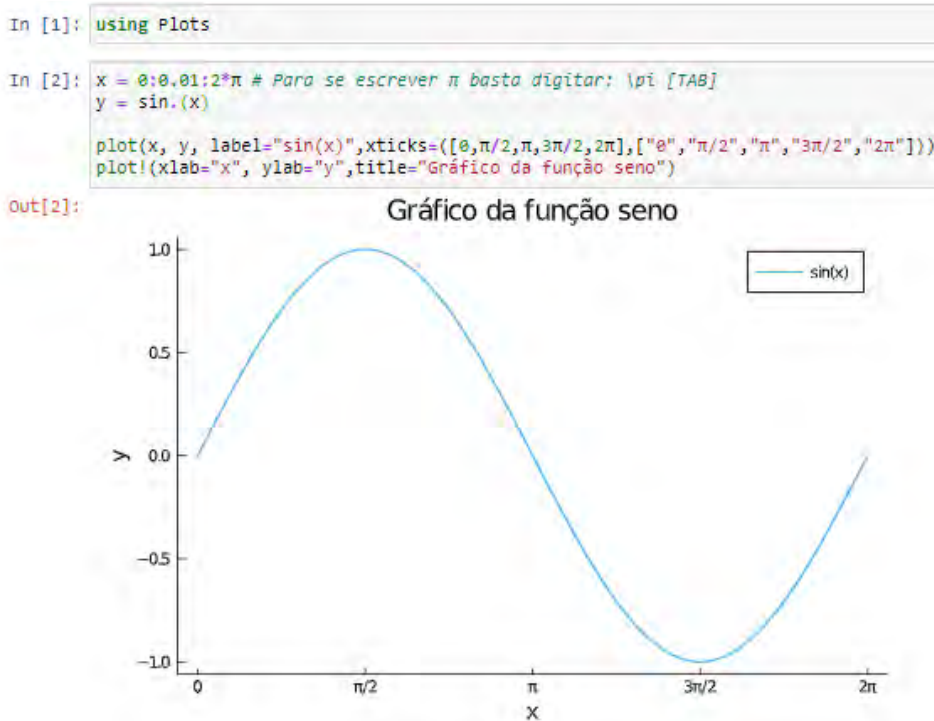
Fonte: Elaborada pelos autores.

Para entender melhor:

1. Em (In[41]:), definimos nosso vetor;
2. Na célula seguinte passamos o vetor **a** como parâmetro para a função seno. Perceba que essa operação nos retornou um erro;
3. Em (In[43]:), aplicamos o conceito de *broadcasting* e a função seno foi aplicada a cada elemento do vetor, como mostra (Out[43]:).

Para plotar o gráfico dessa função, você também deverá utilizar o conceito de *broadcasting*. Veja como fizemos isso na Figura 7.5.

Figura 7.5 – Gráfico da função seno.



Fonte: Elaborada pelos autores.

Observe os nossos passos:

1. Inicialmente carregamos o pacote Plots;
2. Em `In[2]:`, criamos a variável  $x$ , que representa um vetor com elementos dentro do intervalo de  $0$  a  $2\pi$ . Para criar o gráfico definimos a distância entre dois pontos como sendo  $0.01$ . A essa distância damos o nome de passo<sup>1</sup>;
3. Na linha seguinte, definimos  $y$  como sendo  $\sin(x)$ , isso significa que a função seno será transmitida ponto a ponto do vetor  $x$ ;
4. Por fim, na terceira e quarta linha, traçamos o gráfico de  $x$  versus  $y$ , criamos uma legenda (*label*), nomeamos os eixos e o gráfico.

### 7.1.1 Gráficos do Movimento Harmônico Simples

Imagine que você esteja analisando um sistema massa-mola que realiza um movimento harmônico simples (MHS) descrito pelas seguintes funções:

$$x = 0,2.\cos(2.t + \pi) \quad (7.1)$$

$$v = -0,4.\sen(2.t + \pi) \quad (7.2)$$

$$a = -0,8.\cos(2.t + \pi) \quad (7.3)$$

O comportamento de cada uma dessas funções está descrito na Figura 7.6, a seguir.

1. Veja o que acontece com a curva do gráfico com um passo de valor maior.

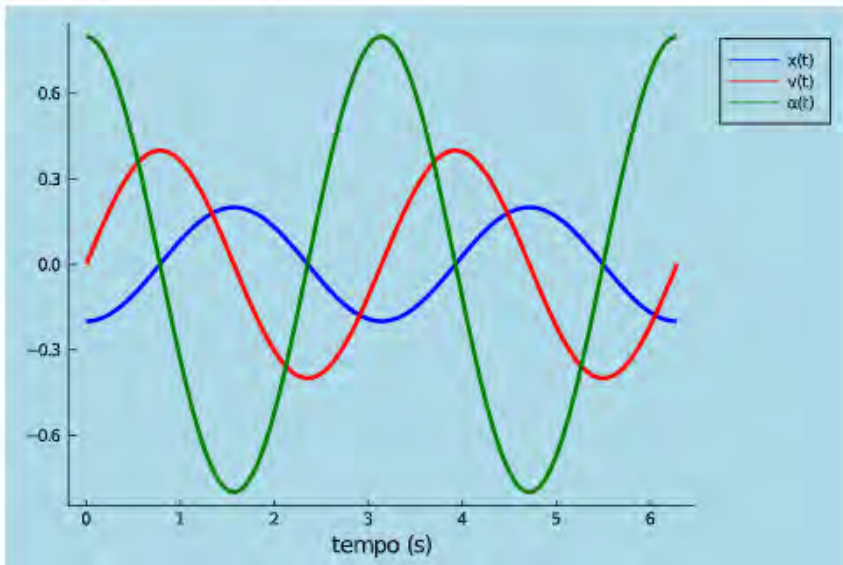
Figura 7.6 – Gráficos da posição, da velocidade e da aceleração em função do tempo para um movimento harmônico simples.

```
In [1]: using Plots
```

```
In [2]: t = range(0,2π,length=100) #Para se escrever π basta digitar: \pi [TAB]
x = 0.2cos.(2t .+ π)
v = -0.4sin.(2t .+ π)
a = -0.8cos.(2t .+ π) #Para se escrever a basta digitar: \alpha [TAB]

plot(t, x, label="x(t)",color=:blue,lw=:3,background_color=:lightblue)
plot!(t, v, label="v(t)",color=:red,lw=:3)
plot!(t, a, label="a(t)",color=:green,lw=:3)
plot!(xlab="tempo (s)")
plot!(legend=:outertopright)
```

Out[2]:



Fonte: Elaborada pelos autores.

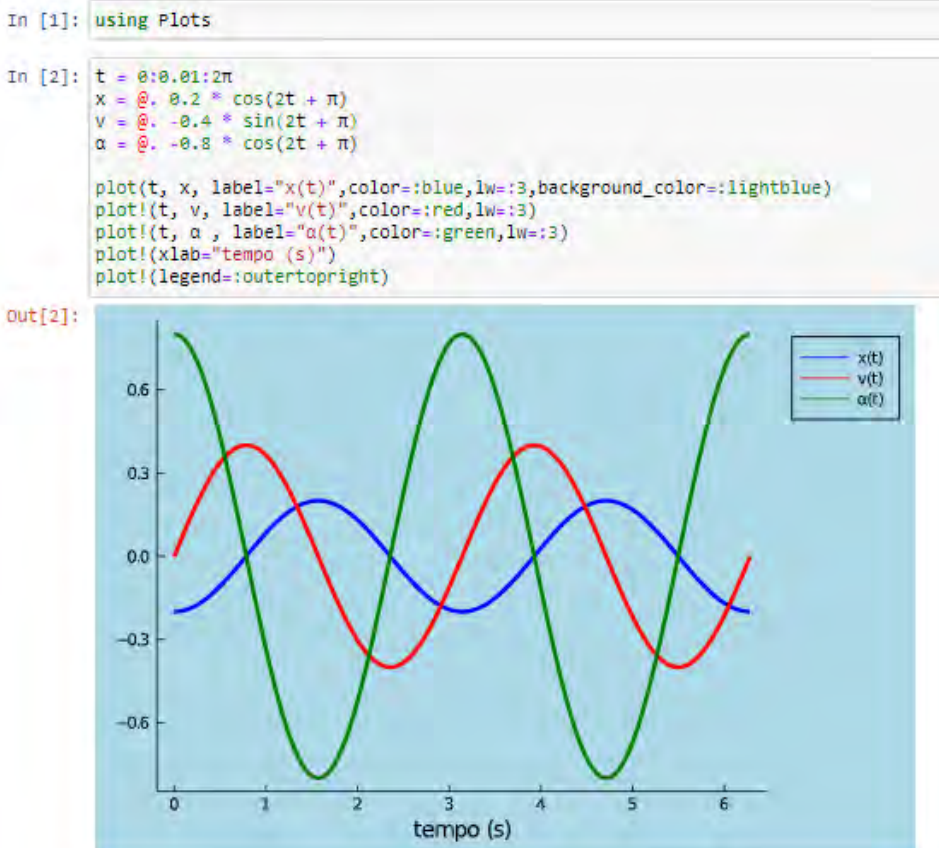
Vejamos o que esse código difere dos anteriores:

1. Em (In[2]:), definimos o vetor tempo (t). Nesse caso, utilizamos a função **range(0, 2π, length=100)** para gerar um vetor que conterà 100 elementos e possuirá um comprimento de 0 a 2π;
2. Na segunda linha do código, definimos a função (x), aqui, representada pela expressão  $0.2\cos.(2t. + \pi)$ ;
3. Nas próximas duas linhas, inserimos as funções v e a;

Aqui, chamaremos sua atenção para o ponto que aparece após o cosseno (cos.), seno (sin.) e antes do sinal de (.+). O primeiro ponto representa o conceito de *broadcasting*, já o segundo permitirá que um vetor seja somado a um escalar.

As outras linhas seguem as mesmas orientações dadas no capítulo 3. Outra sintaxe para plotar este mesmo gráfico é apresentada na Figura 7.7.

Figura 7.7– Gráficos da posição, da velocidade e da aceleração em função do tempo para um movimento harmônico simples (outra opção de código).



Fonte: Elaborada pelos autores.

Note que nessa nova sintaxe, aparece um (@.) na frente das funções. Ele tem a função de inserir, automaticamente, um ponto entre todos os operadores e chamadas da função.

Caso queira saber mais sobre o (@.), tecle **?@.** no jupyter e, em seguida, clique em **Run** para que Julia passe mais informações.

Agora, chegou a sua vez de experimentar o conceito de *broadcasting* apresentado neste capítulo.





No início do capítulo 2, falamos que Julia leria seu código linha por linha, de cima pra baixo e da esquerda para a direita. Pois bem, isso não é totalmente verdadeiro. Nem sempre todas as linhas de um programa precisarão ser lidas. Muitas vezes, é melhor decidir qual parte do código será executado com base em uma condição.

Neste capítulo, apresentaremos três formas de se tomar decisões por meio dos comandos **if/elseif/else**, do operador ternário (?) e dos operadores lógicos. Vamos conhecer cada um deles?



## 8.1 if/elseif/else

Nesta seção, entenderemos como a sintaxe **if/elseif/else** funciona por meio do exemplo apresentado na Figura 8.1.

Figura 8.1 – Estrutura condicional if/elseif/else.

```
In [*]: println("Qual a temperatura do café?")
        temperatura=parse(Int64,readline())
        if temperatura>80
            println("Muito quente, cuidado!")
        elseif 60 <= temperatura <= 80
            println("Temperatura ideal, aprecie!")
        else
            println("Não tome, muito frio!")
        end
```

stdin>

Qual a temperatura do café?

Fonte: Elaborada pelos autores.

Vamos te explicar linha a linha desse código:

1. Na primeira linha, temos a função *println*. Como já vimos, sua função é imprimir na tela a *string*, que está entre aspas e dentro dos parênteses;

2. Em seguida, criamos a variável *temperatura* e chamamos a função *parse( )* com duas informações: *Int64* e a expressão *readline( )*;

3. Na terceira, quinta e sétima linha, inserimos as seguintes condições: **if temperatura>80**, **elseif 60 <= temperatura <= 80** e **else**. Perceba que se a temperatura for maior que 80 o programa retornará a seguinte string: **Muito quente, cuidado!**. Caso a temperatura esteja entre 60 e 80, o programa imprimirá a *string*: **Temperatura ideal, aprecie!**. Se nenhuma dessas condições for



atendida, o programa imprimirá a *string*: **Não tome, muito frio!**. Cada condição será avaliada até que uma seja verdadeira, para que, então, seu bloco de código correspondente seja executado;

4. Por fim, na nona linha, finalizamos o código com **end** e em seguida clicamos em **Run**.

Ao clicar em **Run**, a função `readline()` será executada e uma caixa de diálogo, com o nome `stdin>`, estará disponível para que você possa indicar um valor para a temperatura do café. Por sua vez, a função `parse()` converterá esse valor digitado em um `Int64`.

Caso queira obter entradas decimais, você poderá mudar `Int64` para `Float64` na função `parse()`. Na Figura 8.2 testamos o nosso código.

Figura 8.2 – Estrutura condicional `if/elseif/else`.

```
In [1]: println("Qual a temperatura do café?")
         temperatura=parse(Int64,readline())
         if temperatura>80
             println("Muito quente, cuidado!")
         elseif 60 <= temperatura <= 80
             println("Temperatura ideal, aprecie!")
         else
             println("Não tome, muito frio!")
         end
```

```
Qual a temperatura do café?
stdin> 70
Temperatura ideal, aprecie!
```

Fonte: Elaborada pelos autores.

Perceba que inserimos a temperatura 70 e o programa nos retornou a *string* *Temperatura ideal, aprecie!*.

Agora chegou a sua vez de testá-lo.

## 8.2 Operador ternário

O operador ternário (?) está relacionado à sintaxe **if-elseif-else**<sup>1</sup>. Ele aparece em diversas linguagens e sempre é usado quando se necessita de trechos de códigos enxutos. Vejamos um exemplo na Figura 8.3.

Figura 8.3 – Sintaxe if/else versus sintaxe do operador ternário.

```
In [1]: if 1<5
        println(12)
        else
        println(-6)
        end
```

12

```
In [2]: (1<5) ? 12 : -6
```

```
Out[2]: 12
```

Fonte: Elaborada pelos autores.

Observe que, em (In[1:]), escrevemos um código com a estrutura if/else. A leitura do código é a seguinte: se 1 for menor que 5, a saída deverá ser 12. Caso contrário, deverá ser -6.

Logo abaixo, em (In[2:]), escrevemos o mesmo código, mas utilizamos a sintaxe do operador ternário. A expressão entre parênteses será analisada e se ela for verdadeira, o primeiro valor será impresso na tela. Se for falso, será impresso o segundo valor.

1. <https://docs.julialang.org/en/v1/manual/control-flow/>

### 8.3 Operadores E (&&) e OU (||)

Em Julia, utilizamos os operadores E (&&) e OU (||) para avaliar se determinadas expressões no código são verdadeiras ou falsas. Vamos entender melhor esses operadores por meio do exemplo apresentado na Figura 8.4.

Figura 8.4 – Exemplos de utilização dos operadores && e ||.

```
In [1]: x=5
        y=3;
Out[1]: 5

In [4]: (x<0) && (x>y)
Out[4]: false

In [5]: (x<0) || (x>y)
Out[5]: true
```

Fonte: Elaborada pelos autores.

Passeando pelas células, veja que:

1. Em (In[1:]), inserimos as variáveis *x* e *y* e atribuímos os valores 5 e 3, respectivamente. Serão esses os valores a que nos remeteremos nas próximas células, quando falarmos de *x* e *y*;

2. Em (In[4:]), criamos as expressões **(*x*<0)** e **(*x*>*y*)** e utilizamos o operador && para uni-las. Esse operador só retornará uma saída com o nome *true* (verdadeiro) se as duas expressões forem verdadeiras, caso contrário, retornará **false** (falsa). No nosso exemplo, a primeira expressão é falsa, no entanto, a segunda é verdadeira. Veja em (Out[4:]) que o programa nos retornou **false**, como esperado;

3. Em (In[5:]), repetimos as expressões apresentadas em

(In[4]:) e utilizamos o operador `||` para uni-las. Esse operador nos retornará uma saída com o nome **true** (verdadeiro), se pelo menos uma expressão for verdadeira. Caso contrário, retornará **false**. No nosso exemplo, como a segunda expressão é verdadeira, o programa nos retornou **true** (verdadeiro).

Agora chegou a sua vez de criar um novo código com esses operadores.





Em Julia, as avaliações repetidas (iterações) apresentam-se por meio dos laços *for* e *while*. Ambos têm a função de repetir um determinado trecho de código, a diferença é que no laço *for*, o número de vezes que o código será repetido é conhecido, diferentemente do *while*. Cada um desses laços será apresentado nas seções deste capítulo.

### 9.1 Laço *for*

Sabemos que o laço *for* é utilizado em situações em que o número de iterações (repetições) é conhecido com antecedência. Normalmente, ele é utilizado quando precisamos processar elementos de um vetor, um a um. Vejamos um exemplo de sua sintaxe na Figura 9.1.



Figura 9.1 – Exemplo 1 – Laço for.

```
In [53]: for i = [ 1 , 2 , 3 ]  
        print( i , " " )  
        end  
  
1 2 3
```

---

Fonte: Elaborada pelos autores.

Observe que na primeira linha do código, inserimos o *for* seguido da variável  $i$ <sup>1</sup>. Essa variável assumirá os valores do vetor [1,2,3]. Na linha seguinte, solicitamos que o programa imprima os valores de  $i$  com espaço entre eles. No código, esse espaço está representado entre aspas. Por fim, finalizamos o laço *for* com um **end**.

Vamos entender o que ocorre dentro do laço?

O *for* repetirá o comando *print* três vezes, uma para cada valor do vetor [1,2,3], da seguinte forma: **i** inicialmente assumirá o valor de 1 e em seguida, o comando *print* imprimirá esse valor na tela. Como ainda há valores no vetor, o fluxo do programa retornará para a primeira linha. Esse processo se repetirá até que todos os valores sejam impressos. Quando isso acontecer, o programa será encerrado

Agora, vamos te dar um exemplo da aplicação desse laço na Física. Vejamos a seguinte situação: uma bola foi lançada verticalmente para cima com velocidade inicial de 3.5 m/s, em um local da superfície terrestre em que podemos considerar a

1. O sinal de igual, logo após o  $i$ , pode ser substituído por **in**.

aceleração gravitacional igual a  $9.81 \text{ m/s}^2$ . Desejamos descobrir a altura máxima atingida por essa bola, bem como plotar seu gráfico. Veja na Figura 9.2 o programa que construímos.

Figura 9.2 – Lançamento vertical usando o laço for.

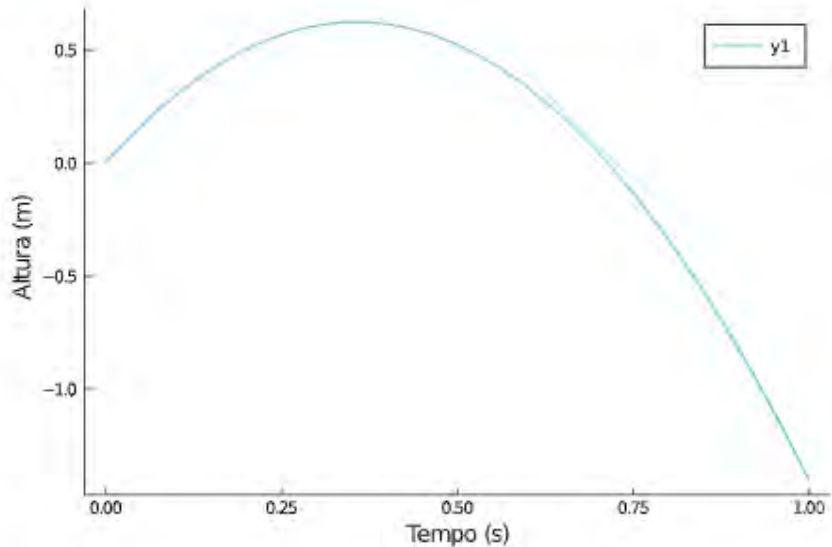
```
In [1]: using Plots
```

```
In [2]: using Printf
Vo = 3.5 # velocidade inicial
g = 9.81 # aceleração da gravidade
t = range(0, 1, length=1000) # 1000 pontos no intervalo
y = @.Vo*t - 0.5g*t^2
altura_maxima=y[1]
for i in range(1, step=1, length(y))
    if y[i] > altura_maxima
        altura_maxima = y[i]
    end
end

println("y(m):",@sprintf("%.3f",altura_maxima))
plot(t,y)
plot!(xlabel="Tempo (s)", ylabel=" Altura (m)")
```

```
y(m):0.624
```

```
Out[2]:
```



Fonte: Elaborada pelos autores.

Observe que:

1. Em (In[1:]), iniciamos o nosso código com *using Plots*;
2. Na primeira linha de (In[2:]), inserimos a função **Printf**;
3. Nas linhas seguintes, indicamos os valores de  $v_0$ ,  $g$  e  $t$ .  
Perceba que  $t$  é um vetor com 1000 pontos entre 0 e 1;
4. Na quinta linha, definimos a função da posição vertical da bola em função do tempo ( $y$ ). O operador  $@.$  será usado porque iremos multiplicar o valor de  $g$ , que é um escalar, por um vetor, que no caso é  $t$ ;
5. Na linha seguinte, indicamos o primeiro valor de  $y^2$  como sendo a *altura\_maxima*. Isso permitirá que o laço *for* tenha um valor inicial para comparar com os outros valores do vetor  $y$ , na busca pela *altura\_maxima*;
6. A partir da sétima linha, indicamos a variável **i** e os valores assumidos por ele durante o laço, que no caso estão representados pela **range(1,step=1, length(y))**. Para maiores informações, consulte a seção 6.1.1;
7. Na linha seguinte, inserimos a condição **if y[i] > altura\_maxima**. Essa condição solicitará a Julia que encontre, dentre os valores assumidos por  $y$ , o valor que corresponde a **altura\_maxima**;
8. Na nona linha, o valor da altura máxima é atualizado;
9. A próxima linha, contendo **end** finaliza-se o **if** e o seguinte, o laço **for**;
10. Nas duas linhas seguintes, solicitamos ao programa que imprima o valor da altura máxima e em seguida, plote o gráfico da altura pelo tempo;

2. A sintaxe  $y[1]$  equivale a  $y_1$ , ou seja, o número 1 representa um índice para  $y$  e não o valor do tempo. Em Julia, os índices sempre se iniciam em 1, diferentemente de outras linguagens em que os índices se iniciam em zero.



11. Por fim, nomeamos os eixos.

Após ter clicado em **Run**, o programa nos retornou a **altura\_maxima** de 0.264 e o gráfico da trajetória da bola.

## 9.2 Laço *while*

Nesta seção, falaremos sobre o laço *while*. Esse laço é utilizado quando desejamos repetir um bloco de código, mas não sabemos quantas iterações (repetições) serão necessárias. Vejamos na Figura 9.3 um exemplo da utilização desse laço em uma contagem regressiva para o lançamento de um foguete.

Figura 9.3 – Exemplo 1 - Laço *while*.

```
In [3]: i=5
while i>0
    println(i)
    i-=1      # i-= 1 é equivalente a sintaxe i = i-1
end

println("LANÇAR!")

5
4
3
2
1
LANÇAR!
```

Fonte: Elaborada pelos autores.

Como sempre fazemos, vamos te explicar linha a linha deste código:

1. Na primeira linha, declaramos a variável **i** e atribuímos a ela o valor 5;
2. Em seguida, iniciamos o laço *while* indicando **i>0**. Esse comando solicitará a Julia que enquanto o valor de **i** for maior que zero, realize as instruções contidas no laço;

3. A primeira instrução do laço, situada na terceira linha, solicita ao programa que imprima o valor de **i**;

4. A segunda instrução solicita a atualização do valor de **i**, subtraindo 1(um) do valor antigo;

5. Por fim, o laço é finalizado com o **end** e solicitamos ao programa que imprima LANÇAR!.

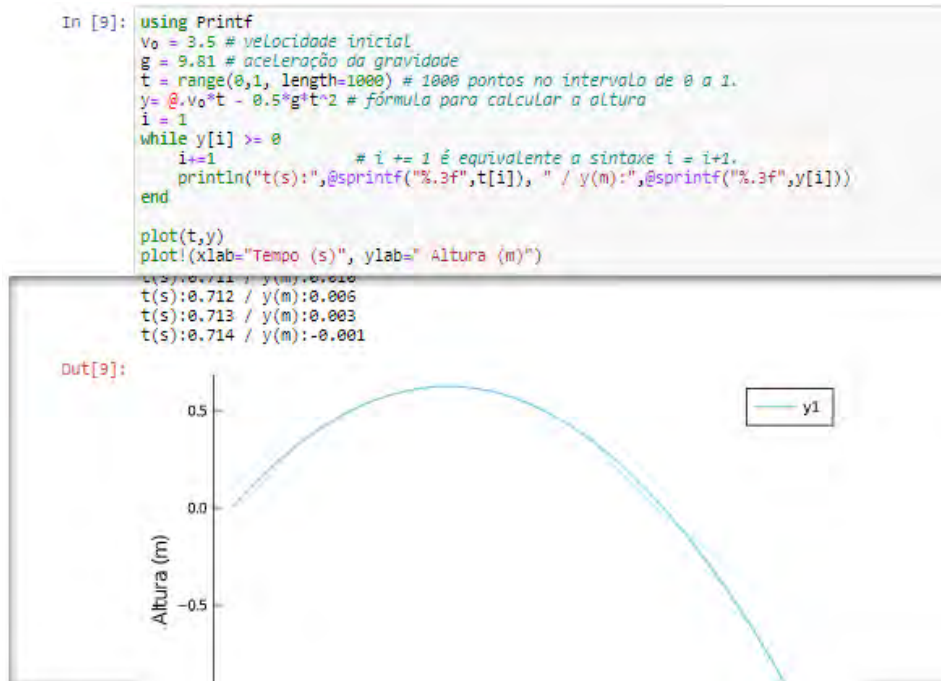
Vamos entender como ocorre o processo dentro do laço *while*?

Em nosso código, o valor inicial de **i** é igual a 5 e temos a condição de ele ser maior que zero. De posse dessas informações, o laço verificará se 5 é maior que zero e, em caso afirmativo, o programa imprimirá o valor 5, para em seguida subtrair uma unidade.

Agora temos o valor 4. O laço irá novamente verificar se 4 é maior que zero, para então imprimi-lo e subtrair deste valor uma unidade. Todo esse processo se repetirá até o valor de **i** se igualar a zero. Quando isso acontecer a condição **i**>0 será falsa, o que fará com que o programa saia do laço e execute a primeira linha fora dele. No nosso exemplo, o programa irá executar a função *println* e imprimirá LANÇAR!.

Vejamos agora, na Figura 9.4, a utilização desse laço na Física, utilizando a mesma situação apresentada para o laço *for*.

Figura 9.4 – Lançamento vertical usando o laço *while*.



Fonte: Elaborada pelos autores.

Vamos explicar linha a linha desse código<sup>3</sup>:

1. As 5 primeiras linhas são as mesmas utilizadas para o programa criado com o laço *for*;
2. Na linha seguinte, declaramos a variável **i**;
3. Na sétima linha, inserimos a condição  $y[i] \geq 0$  para o laço *while*. Ele será executado enquanto essa condição estiver sendo atendida;
4. Nas linhas seguintes, informamos ao Julia que a variável **i** deverá ser atualizada ( $i+=1$ ) e que, a cada atualização, imprima a posição e o tempo que ela levou para atingir essa posição;

3. Estamos considerando que você já tenha importando o pacote Plots para seu código usando **using Plots**.

5. Na décima linha, finalizamos o laço com **end**;

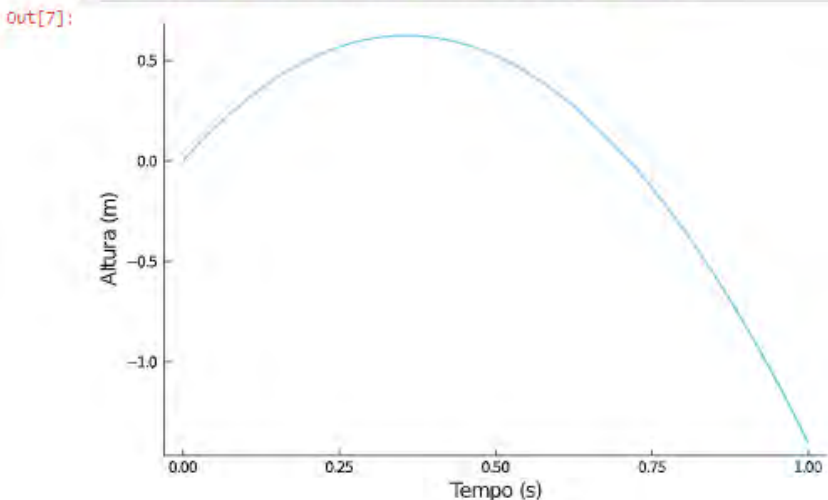
6. Por fim, solicitamos ao Julia que plote o gráfico do movimento e nomeie os eixos.

O processo dentro do laço *while* acontecerá de forma semelhante ao exemplo anterior. Caso queira salvar os valores do tempo e posição da bola em um arquivo CSV, basta seguir as orientações apresentadas no capítulo 4. Vejamos um exemplo de código que realiza essa tarefa na Figura 9.5.

Figura 9.5 – Utilizando os pacotes DataFrames e CSV para salvar os dados do movimento.

```
In [6]: using DataFrames
        using CSV
        using Plots
```

```
In [7]: using Printf
        vo = 3.5 # velocidade inicial
        g = 9.81 # aceleração da gravidade
        t = range(0,1, length=1000) # 1000 pontos no intervalo de 0 a 1.
        y = @. vo*t - 0.5*g*t^2 # fórmula para calcular a altura
        i = 1 # declarando uma variável global
        while y[i] >= 0
            i+=1 # i += 1 é equivalente a sintaxe i = i+1.
        end
        df = DataFrame(tempo=t[i], posição=y[i])
        CSV.write("C:\\Users\\adeil\\Downloads\\lançamento.csv", df)
        plot(t, y, xlabel="Tempo (s)", ylabel="Altura (m)", legend=false)
```



Fonte: Elaborada pelos autores.

A diferença deste último gráfico para o anterior está nas 3 linhas após o **end**. Vejamos:

1. Criamos a variável `df`. Esta variável é um `DataFrame` que guardará os valores do tempo e da posição do movimento da bola;

2. Na próxima linha salvamos o `DataFrame` em um arquivo CSV usando a função **`CSV.write`**;

3. Finalmente solicitamos ao Julia que plote o gráfico do movimento.

Ao clicarmos em **Run**, o arquivo CSV será salvo na pasta indicada e o gráfico do movimento será plotado.



100



# OUTPUT



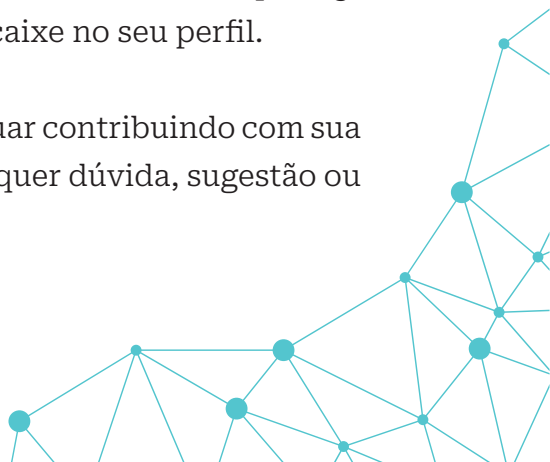
Se você chegou até aqui tendo passado por todos os capítulos, parabéns! Esperamos que tenha sido uma jornada de muito aprendizado e que nosso objetivo, que era o de introduzir você no universo da linguagem de programação Julia, tenha sido alcançado.

Agora, chegou a hora de você caminhar sozinho. Nessa caminhada, você poderá se aventurar pelo mundo da computação paralela, do aprendizado de máquina e da inteligência artificial, por que não? Existe um universo em Julia a descobrir.

Caso não se sinta preparado, o site Julia Academy<sup>1</sup> possui diversos cursos livres. Ele foi preparado pelos principais desenvolvedores do Julia em colaboração com *Julia Computing*. Vá lá e descubra algum curso que se encaixe no seu perfil.

Ficaremos muito felizes em continuar contribuindo com sua formação ou saber de seu sucesso. Qualquer dúvida, sugestão ou

1. <https://juliaacademy.com/>



comentários, basta enviar um e-mail para: [juliacomfisica@gmail.com](mailto:juliacomfisica@gmail.com).

Abraços!



# BIBLIOGRAFIA



Aurelio Amerio. *From zero to julia!*, 2020. URL <https://techtok.com/from-zero-to-julia/>. Último acesso em 9 de outubro de 2020.

Amaury B. André. *Aprendendo Julia: Introdução à linguagem de programação Julia*. Amaury Bosso André, 2020a.

Amaury B. André. *Aprendendo Julia: Introdução à DataFrames e Ciência de Dados em Julia*. Amaury Bosso André, 2020b.

Tanmay Bakshi. *Tanmay Teaches Julia for Beginners: A Springboard to Machine Learning for All Ages*. McGraw-Hill, 2020.

Ivo Balbaert. *Julia 1.0 Programming: Dynamic and high-performance programming to build fast scientific applications*. Packt, 2018.



Wolfgang Bauer, Gary D Westfall, and Helio Dias. *Física para Universitários-Relatividade, Oscilações, Ondas e Calor*. AMGH Editora, 2013.

Julia Computing. *Julia academy*, 2020. URL <https://juliaacademy.com/>. Último acesso em 13 de outubro de 2020.

Allen Downey and Ben Lauwens. *Think Julia: How to Think Like a Computer Scientist*. O'Reilly, 2018.

Adeil Araújo e Meirivâni Meneses. *Julia com física*, 2020. URL <https://juliacomfisica.github.io/>. Último acesso em 16 de dezembro de 2020.

Gerson J Ferreira. *Introduction to Computational Physics*. Gerson J Ferreira, 2016.

LO Man Hei. *Julia tutorial*, 2019. URL <https://bit.ly/2IhIW8J>. Último acesso em 9 de outubro de 2020.

Gelson Iezzi and Samuel Hazzan. *Fundamentos de matemática elementar, 4: seqüências, matrizes, determinantes, sistemas*. Atual, 2004.

Paulo Jabardo. *Introdução à programação em julia*, 2020. URL <http://vento.eng.br/posts/intro-to-julia/>. Último acesso em 10 de outubro de 2020.

Francisco Ramalho Junior, Nicolau Gilberto Ferraro, and Paulo Antonio de Toledo Soares. *Os fundamentos da física*. Moderna, 2007.

Julia Programming Language. *Julia 1.5 documentation*, 2020. URL <https://docs.julialang.org/en/v1/>. Último acesso em 18 de outubro de 2020.

Svein Linge and Hans Petter Langtangen. *Programming for Computations-Python: A Gentle Introduction to Numerical Simulations with Python* 3.6. Springer Nature, 2020.

Abel Siqueira. *Tutoriais de julia*, 2020. URL <https://bit.ly/33LS5yw>. Último acesso em 11 de outubro de 2020.

Gregor Steele. *Vidanalysis: An app for physical analysis of motion in videos, how to*, 2015. URL <http://vidanalysis.com/howto/>. Último acesso em 10 de outubro de 2020.

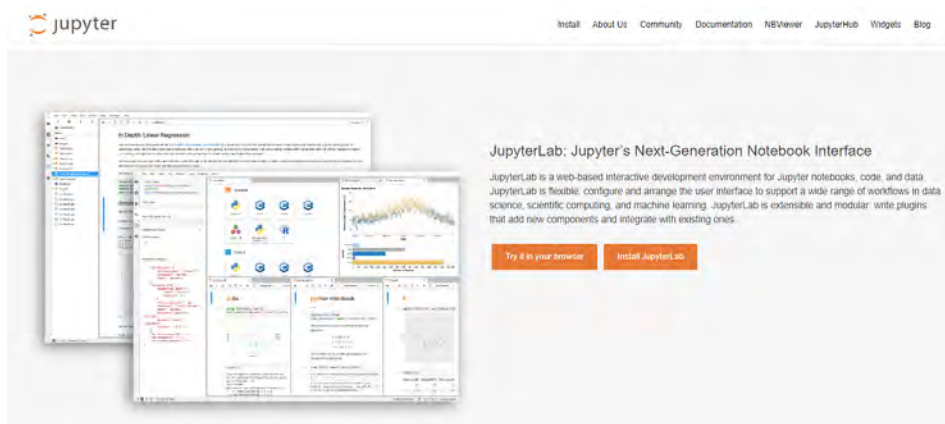




Neste apêndice, vamos orientar você em seu primeiro acesso ao site [jupyter.org](https://jupyter.org), para que você possa programar em Julia de forma *online*. Para isso, siga os seguintes passos:

1. Abra uma nova aba em seu navegador e digite <https://jupyter.org/>. A página que abrirá, terá a aparência da Figura A.1;

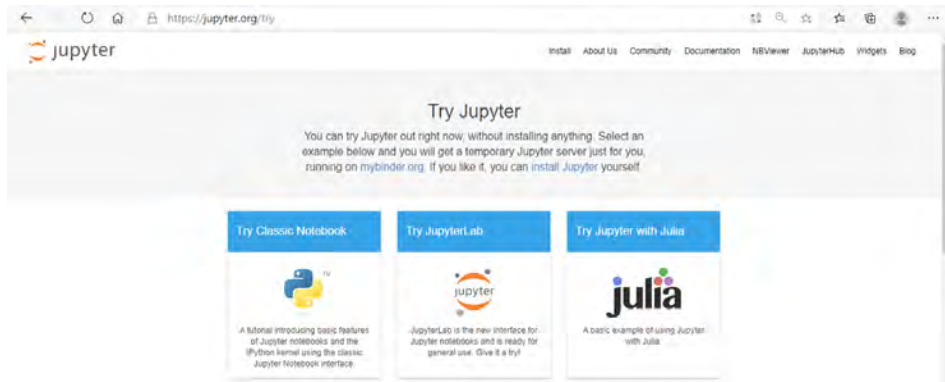
Figura A.1 – Acesso ao Julia pelo Jupyter online - Site Jupyter.org.



Fonte: Elaborada pelos autores.

2. Clique em *Try it in your browser* (experimente em seu navegador) para ser redirecionado para a página apresentada na Figura A.2;

Figura A.2 – Acesso ao Julia pelo Jupyter online - Experimentar Julia no Jupyter Online

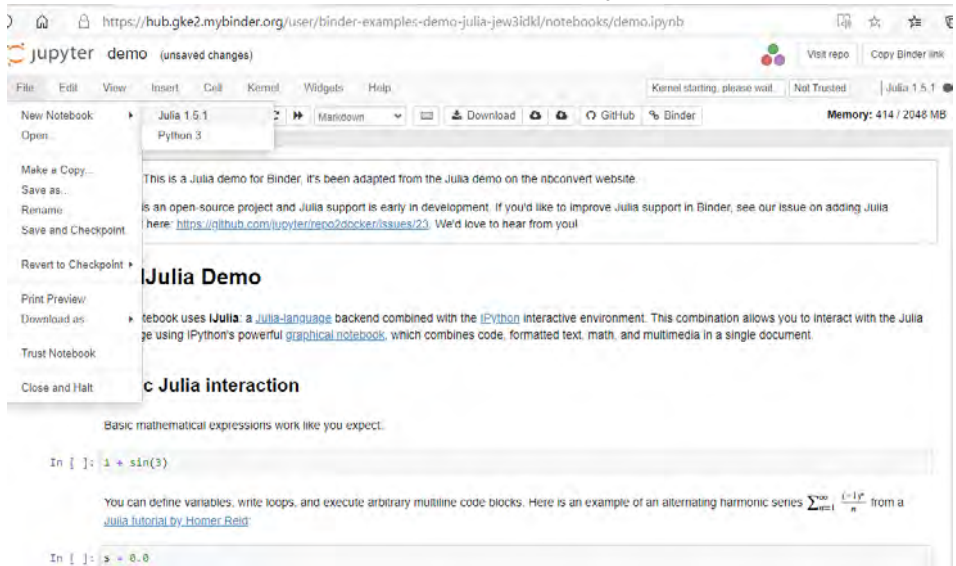


Fonte: Elaborada pelos autores.

3. Em seguida, escolha a opção *Try Jupyter with Julia* (experimente Jupyter com Julia);

4. No menu File (Arquivo), clique em *New notebook* (Novo caderno) para abrir um Jupyter em branco, como mostra a Figura A.3;

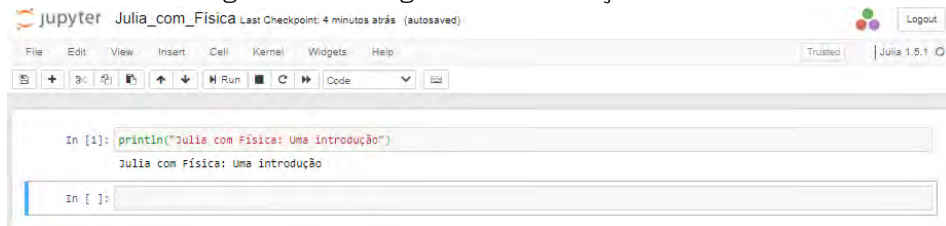
Figura A.3 – Página de demonstração do IJulia.



Fonte: Elaborada pelos autores.

5. Insira na célula de entrada a expressão `println("Julia com Física: Uma introdução")`, em seguida, clique em **Run** para Julia executá-la. Veja esse exemplo na Figura A.4.

Figura A.4– Código de demonstração no IJulia.



Fonte: Elaborada pelos autores.

Agora chegou a sua vez de inserir outras expressões. Não se preocupe, você não quebrará nada!





# OBTENDO UM ARQUIVO CSV A PARTIR DA VIDEO- ANÁLISE DO MOVIMENTO DE UM OBJETO

Apêndice



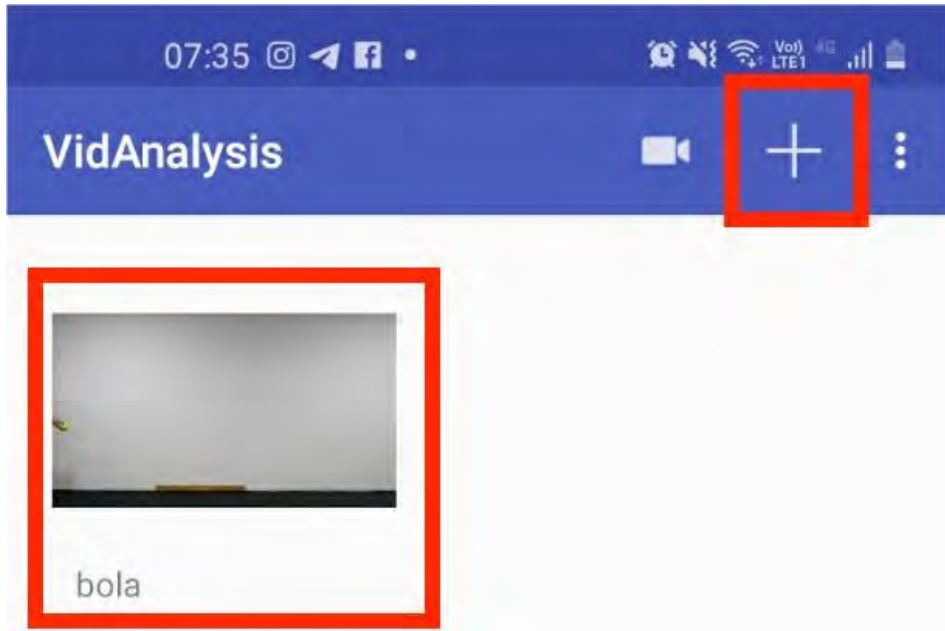
## B

O objetivo deste apêndice é iniciar você de forma rápida ao processo de análise de vídeo de um determinado movimento. Para isso, vamos utilizar o aplicativo *VidAnalysis*, que está disponível para *smartphones* com o sistema operacional *Android*.

Uma vez instalado o aplicativo, abra-o e clique no ícone +, situado no canto superior direito da sua tela, para carregar um vídeo de um movimento qualquer. Como mostra a Figura B.1.



Figura B.1- Tela inicial do aplicativo VidAnalysis.



Fonte: Elaborada pelos autores.

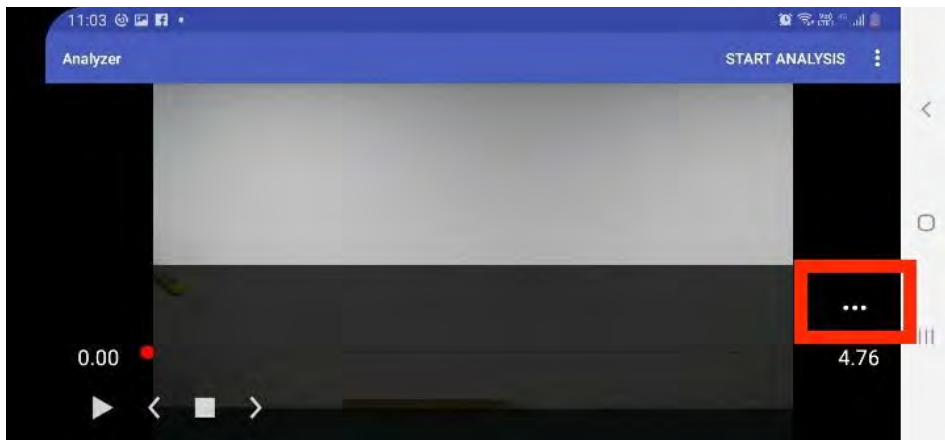
É importante destacar que a análise do vídeo deve ser feita na mesma posição em que foi gravado, ou seja, se o vídeo foi gravado no formato paisagem, teremos que analisá-lo também nesse formato. Outro ponto de atenção é que no vídeo deverá existir algum objeto de tamanho conhecido, como por exemplo, uma régua, para servir de referência. Em nosso vídeo, estamos utilizando uma régua de 1 metro.

Após escolher o vídeo, clique na opção *START ANALYSIS*, que aparece no canto superior direito da tela do *smartphone* e em seguida clique nas extremidades do objeto de tamanho conhecido, para inserir seu tamanho em metros.

Após se inserir o tamanho do referencial, surge na tela os eixos x e y, para que você escolha a origem do movimento do objeto. Confirme a origem escolhida, clicando no ícone **V**, no canto superior direito da tela. É importante que a origem do eixo coincida com a origem do movimento.

Para visualizar o botão *play*, o controle deslizante vermelho e os ícones < >, basta clicar nos três pontos que aparecem no canto inferior direito da tela. Você poderá utilizá-los para avançar através do vídeo até o ponto em que o objeto começa a se mover, como mostra a Figura B.2.

Figura B.2 – Ícones úteis durante o processo de análise do vídeo.



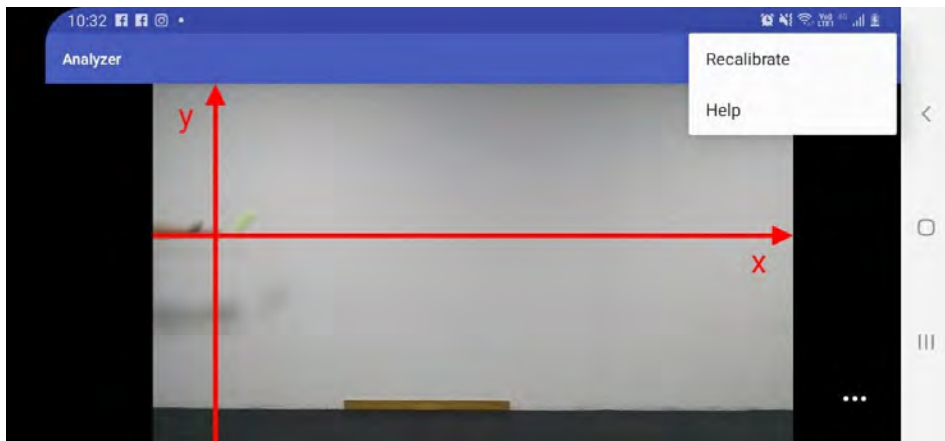
Fonte: Elaborada pelos autores.

Agora, ao clicar no objeto que você está analisando, uma cruz azul aparecerá e o vídeo avançará um quadro automaticamente. A cada novo clique, o vídeo irá avançando até o objeto parar.

Para que os dados gerados representem o movimento real

do objeto, é extremamente importante que os pontos marcados durante a trajetória da bola sejam o mais próximo possível da posição do objeto no vídeo. Se você não estiver satisfeito com a posição de suas marcações, clique nos três pontinhos no canto superior direito da tela e recalibre o processo, como mostra a Figura B.3.

Figura B.3– Recalibrando o movimento.

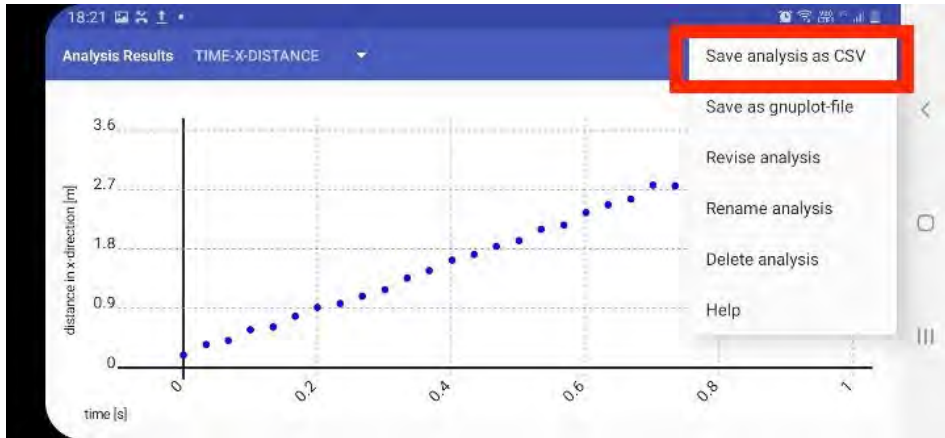


Fonte: Elaborada pelos autores.

Após rastrear todos os pontos, toque no ícone do disquete no canto superior direito da tela, insira um nome para sua análise e clique em ok.

O aplicativo, então, fornecerá alguns gráficos e uma tabela que você poderá exportar no formato CSV. Para exportá-la, basta clicar nos três pontinhos, no canto superior direito, e em seguida clicar em *SAVE ANALYSYS AS CSV*, como mostra a Figura B.4.

Figura B.4 – Obtendo um arquivo de dados no formato CSV.



Fonte: Elaborada pelos autores.

Agora convidamos você a gravar seu próprio vídeo, importar a tabela de dados para o Julia e personalizar seus próprios gráficos.



# **CEARÁ**

**GOVERNO DO ESTADO**

SECRETARIA DA EDUCAÇÃO