Universidade de Federal de Catalão
Instituto de Física

Petrus Henrique Ribeiro dos Anjos

Métodos Computacionais em Física Básica: Da Física I a Física IV com *Python*

Em Construção! Texto Não Revisado.

Catalão

2021

Prefácio

If you want to master something, teach it.
Richard Feynman

A idéia central deste material didático é servir como um complemento as disciplinas de Física Básica (normalmente chamadas de Física I a Física IV) abordando como utilizar computação em Física. É portanto uma disciplina diferente das disciplinas "Física Computacional" que tem aparecido nos currículos das graduações em Física pelo Brasil. A ideia é que desde o início da sua graduação seja natural para o futuro Físico empregue computação nas suas atividades. Na UFCAT, isso é feito via 4 disciplinas que acontecem em paralelo com as disciplinas de Física Básica. Em princípio, os alunos já cursaram as disciplina "Algorítimos e Programação de Computadores".

Os capítulos apresentam problemas de Física em ordem semelhante a livros tradicionais como [1] e [2, 3, 4, 5]. Tópicos a respeito de métodos computacionais foram colocados nos apêndices e devem muito a referências que já utilizam Python como [6, 7]. Também haverá um apêndice tratando do uso de LaTeX.

Os cursos geralmente poderão começar pelo capítulo 1 e utilizar alguns dos apêndices. No demais as ementas da UFCAT sugerem a seguinte disposição:

1. **Métodos Computacionais em Física 1:** Capítulos 2, 3, 4, 5, 6. No fluxo normal a disciplina de Física 1 é feita no segundo período. Os alunos estarão cursando as disciplinas "Física 1" e "Cálculo Numérico" em paralelo a esta disciplina.

2. **Métodos Computacionais em Física 2:** Capítulos 7, 8, 9, 10, 11. No fluxo normal a disciplina de Física 1 é feita no terceiro período. Os alunos estarão cursando em paralelo as disciplinas "Física 2", "Probabilidade e Estatística" e "Cálculo Vetorial".

3. **Métodos Computacionais em Física 3:** Capítulos 12, 13, 14. No fluxo normal a disciplina de Física 1 é feita no quarto período. Os alunos estarão cursando em paralelo as disciplinas "Física 3", "Cálculo III" (equações diferenciais ordinárias) e "Física Matemática 1" (o que permite uso mais extensivo de números complexos).

4. **Métodos Computacionais em Física 4:** Capítulos 15, 16, 17. No fluxo normal a disciplina de Física 1 é feita no quinto período. Os alunos estarão cursando em paralelo a disciplina "Física 4".

O material esta sendo escrito em paralelo ao período em que ministro as disciplinas e portanto esta sendo escrito de maneira não sequencial, obedecendo a demanda de oferta do IF/UFCAT. A tabela abaixo mostra a estimativa para conclusão de cada capítulo:

Capítulo	Título	ETA
1	Introdução ao Python	09/05/2022
2	Cinemática	10/10/2022
3	Dinâmica	17/10/2022
4	Trabalho e Energia	24/10/2022
5	Momento Linear	31/10/2022
6	Rotações, Torque e Momento Angular	17/10/2022
7	Gravitação	16/05/2022
8	Oscilador Harmônico	23/05/2022
9	Ondas	30/05/2022
10	Fluidos	06/06/2022
11	Termodinâmica e Teroia Cinética dos Gases	13/06/2022
12	Campo Elétrico	10/10/2022
14	Campo Magnético	24/10/2022
15	Equações de Maxwell e Oscilações Eletromagnética	07/11/2022
16	Óptica Física e Geométrica	21/11/2022
17	Princípio de Fermat e Outros Problemas de Extremização	05/12/2022
A	Elementos de Python	20/06/2022
B	Phyton em Aplicações Numéricas: Numpy	27/06/2022
C	Construção de Gráficos com Matplotlib	04/07/2022
D	Computação Algébrica com Sympy	11/07/2022
E	Elementos de Cálculo Numérico	18/07/2022
F	Usando LaTeX	25/07/2022

Tabela 1: Cronograma previsto para a produção dos Capítulos.

Sumário

Prefácio **i**

1 Introdução ao Python **1**

1.1 Instalação . . . 3
1.2 Primeiros Passos no Jupyter Notebook . . . 4
1.3 Operações Matemáticas Simples . . . 10
1.4 O "Bê-a-bá" sobre variáveis. . . . 13
1.4.1 Nomeando Variáveis . . . 15
1.4.2 Alguns tipos de variáveis . . . 15
1.5 Instruções de Controle . . . 20
1.5.1 Condicionais . . . 21
1.5.2 if ... elif ... else . . . 21
1.5.3 while . . . 22
1.5.4 for . . . 24
1.6 Funções . . . 26
1.7 Bibliotecas: o Básico . . . 27
1.8 Gráficos Simples . . . 29
1.9 Lendo dados de arquivos . . . 29
1.10 Exercícios . . . 29

2 Cinemática **33**

3 Dinâmica **35**

4 Trabalho e Energia **37**

5 Momento Linear **41**

6 Rotações, Torque e Momento Angular **43**

7 Gravitação **45**

8 O Oscilador Harmônico **49**

9 Ondas **53**

10 Fluidos **55**

11 Termodinâmica e Teoria Cinética dos Gases **57**

12 O Campo Elétrico **61**

13 Circuitos Elétricos **63**

14 O Campo Magnético **65**

15 Equações de Maxwell e Oscilações Eletromagnéticas **67**

16 Óptica Física e Geométrica **69**

17 Princípio de Fermat e Outros Problemas de Extremização **71**

A Elementos de *Python* **75**

A.1 Mais informações sobre tipos de variáveis 75

A.2 Programação Orientada a Objetos em Python 75

B *Python* em aplicações numéricas: Numpy **77**

B.1 Arrays 77

B.2 Métodos Estatísticos 77

B.3 Polinômios 77

B.4 Álgebra Linear 77

B.5 Sorteios Aleatórios 77

C Construção de Gráficos com Matplotlib **79**

C.1 Mais obções básicas com gráficos 79

C.2 Histogramas 79

C.3 Plots de Contorno 79
C.4 Gráficos em 3D 79
C.5 Animações 79

D Computação Algébrica com Sympy **81**

E Elementos de Cálculo Numérico **83**
E.1 Soluções Numéricas 83
E.2 Integração Numérica 83
E.3 Diferenciação Numérica 83
E.4 Equações Diferenciais Ordinárias 83
E.4.1 Método de Euler 83
E.4.2 Variações do Método de Euler 83
E.4.3 Método de Runge-Kutta 83
E.5 Método de Monte Carlos 83
E.6 Métodos Estocásticos 83
E.7 Equações Diferenciais Parciais 83
E.7.1 Equação de Laplace 83
E.7.2 Equação da Onda 83
E.7.3 Equação do Calor 83

F Usando LaTeX **85**

Lista de Códigos Computacionais

1.1 Determinando se um ano é bissexto. 22
1.2 Testando a conjectura de Collatz 23
1.3 Testando se um número é primo 24
1.4 Sequência de Fibonacci e a Razão Áurea 25
4.1 Plot dos dados disponíveis no arquivo Potential.txt 37
4.2 Análise dos derivadas do potencial dados disponíveis no arquivo Potential.txt 37
4.3 Energia Máxima de uma partícula presa pelo potencial. Dados disponíveis no arquivo Potential.txt 38
4.4 Região permitida para a partícula presa pelo potencial. Dados disponíveis no arquivo Potential.txt 39
5.1 Determinando Centro de Massa de uma distribuição aleatória em 2D . 41
7.1 Altitude de um Satélite 45
7.2 Orbitas: Problema de Kepler para 2 Corpos 46
8.1 Oscilador Harmônico Simples 49
8.2 Oscilador Harmônico Duplo 50
8.3 Rotores acoplados 51
11.1 Difusão da Molécula de Um Gás em 2D 58
15.1 Equação de Laplace 67
17.1 Tempo de Viagem na braquistócrona e em outras curvas 71
B.1 Estimando o valor de π usando sorteios aleatórios 77
B.2 Histogram da Distribuição de Probabilidade da caminhada Aleatória . 78

Capítulo 1

Introdução ao Python

Python é uma poderosa linguagem de programação de propósito geral desenvolvida por Guido van Rossum[1] em 1989. É classificada como uma linguagem de programação de alto nível na medida em que lida automaticamente com as operações mais fundamentais (como gerenciamento de memória) realizada no nível do processador ("código de máquina"). É considerado um nível superior linguagem do que, por exemplo, C++, por causa de sua sintaxe (que é próxima de linguagem natural, no caso o inglês) e rica variedade de estruturas de dados nativas, como listas, tuplas, conjuntos e dicionários (vamos discutir um pouco delas no Apêndice A.1). Na verdade o Python, como outras linguagens de alto nível, é um interpretador. Em uma linguagem interpretada como o Python, cada comando é analisado e convertido "ao longo do caminho". Esse processo torna a linguagem interpretada significativamente mais lenta; mas a vantagem é que elas são mais simples para se programar ajustar e depurar porque você não tem para recompilar o programa após cada alteração.

O Python apresenta algumas vantagens:

1. Sua sintaxe limpa e simples torna a escrita de programas Python rápida, simples e geralmente minimiza as chances para que os bugs se infiltrem.

2. É gratuito – Python e suas bibliotecas associadas são gratuitas e de código aberto, ao contrário de ofertas comerciais como Mathematica e MATLAB.

3. Suporte multiplataforma: Python está disponível para todos os com- computador,

[1]Quando começou a implementar o primeiro interpretador Python, Guido van Rossum também estava assistindo a série "Flying Circus" do famoso grupo de comédia Inglês "Monty Python". Van Rossum pensou que precisava de um nome que fosse curto, único e um pouco misterioso, então decidiu chamar a linguagem de Python

incluindo Windows, Unix, Linux e macOS.

4. O Python possui uma grande biblioteca de módulos e pacotes que estendem sua funcionalidade. Isso é ferramentas já prontas que nos permitem realizar uma diversidade de operações complexas comuns como as de álgebra linear, derivação, integração, operações matemáticas simbólicas, construção de gŕaficos. Muitos deles estão disponíveis como parte da "Biblioteca Padrão" fornecida com o próprio Python. Outros, incluindo o NumPy, SciPy, Matplotlib (algumas dessas estão discutidas nos Apêndices) e bibliotecas pandas usadas em computação científica, podem ser baixadas gratuitamente.

5. Python é relativamente fácil de aprender. A sintaxe e as expressões idiomáticas usadas para operações básicas são aplicados de forma consistente no uso mais avançado da linguagem. Mensagens de erro geralmente são avaliações significativas do que deu errado, e não os alertas genéricos de "travamentos" que podem ocorrer em linguagens compiladas de baixo nível.

6. Python é flexível: muitas vezes é descrito como uma linguagem "multi-paradigma" que contém os melhores recursos dos paradigmas procedimentais, orientados a objetos e funcionais.

É claro que o Python também possui algumas desvantagens:

1. A velocidade de execução de um programa Python não é tão rápida linguagens de mais baixo nível como C++ e Fortran. Para aplicações numéricas pesadas, algumas bibliotecas como o NumPy e o SciPy minimizam essa perda de velocidade pois elas mesmas são códigos C++ escondidos sobre uma "tradução" para Python.

2. Para fornecer simplicidade a linguagem, o Python usa quantidade de memória maior. Isso pode ser uma desvantagem ao criar aplicações que demandem um uso otimizado de memória. Por essa razão e pela menor velocidade, o Python é pouco utilizado para aplicativos móveis e para servidores.

3. Python é uma linguagem dita dinâmica. Com isso uma variável que em um dado momento do código é um número inteiro, em outro ponto do código pode ser tornar uma string (e.g. um texto). Isso se chama *Mutabilidade* e pode gerar alguns erros. Isso requer que códigos mais complexos sejam bastante testados.

Em resumo, Python é uma linguagem de programação moderna e bem projetada, ao mesmo tempo fácil de aprender e muito poderosa. Ou seja embora seja simples o suficiente para permitir que estudantes sem experiência prévia em programação resolvam problemas já início da graduação, O Python também é poderoso o suficiente para ser usado para trabalho numérico em física. Na próxima seção, discutimos como instalar o Python em um sistema operacional Linux.

1.1 Instalação

Verifique se já tem o Python instalado, se você usa GNU/Linux, provavelmente já possui alguma versão do Python instalada por padrão. Para conferir, digite em um terminal:

```
$ which python3
```

O Python 3 é a versão atual do Python[2], por ser o futuro da linguagem e pelo fato de sua versão anterior estar em processo de descontinuação. Caso o Python esteja instalado, o sistema vai retornar algo como

```
/usr/bin/python
```

Caso contrário, o sistema retornará algo do tipo

```
which: no python
in (/usr/local/sbin:/usr/local/bin:/usr/bin:/usr...)
```

Caso o Python não esteja instalado podemos instala-lo via um gerenciador de pacotes da distribuição do Linux instalada na máquina. Os gerenciadores de pacotes mais comuns são apt-get que funcionam em distribuições como o Debian, o Ubuntu e o Mint. Para instalar o Python usando o apt-get digite em um terminal:

```
$ sudo apt-get install python3
```

É comum que no desenvolvimento de projetos Python, precisemos instalar diversas bibliotecas para diferentes necessidades. Essas bibliotecas podem ser instaladas manualmente, mas o processo pode ser complicado. Para contornar esse problema, o Python possui uma ferramenta para gerenciamento de pacotes chamado PIP. Isso em geral não é necessário em distribuições Linux como Debian, Ubuntu e Mint. Para instalar o gerenciador de pacotes pip, digite em um terminal:

```
$ sudo apt-get install python3-pip
```

[2] O Python 2 foi descontinuado em 2020.

Caso sua distribuição utilize um gerenciador de pacotes diferente deste (ou você utilize Windows) **não entre em pânico**, na internet há instruções passo a passo para instala-lo.

Para ver a versão do Python que você tem instalado digite no terminal

```
$ python3 --version
```

Após a instalação você poderá usar o Python diretamente do terminal. Por exemplo, digite

```
$ python
$ 6*7
$ a=10**3
$ print(a)
$ quit()
```

É claro seria muito inconivente escrever um código diretamente no terminal. Precisamos agora instalar uma IDE (ambiente de desenvolvimento integra, do inglês *Integrated Development Environment*). Existem várias IDE para Python, durante o curso utilizaremos o Jupyter-Notebook. Para instala-la digite no terminal:

```
$ sudo apt-get install jupyter-notebook
```

Para iniciar o notebook digite no terminal o seguinte comando:

```
$ jupyter-notebook
```

O Jupyter Notebook deverá abrir em seu browser de internet.

1.2 Primeiros Passos no Jupyter Notebook

Quando o Jupyter Notebook abrir em seu navegador você verá o Dashboard do Jupyter Notebook, o qual irá exibir uma lista dos notebooks, arquivos e subdiretórios no diretório onde o servidor do notebook foi inicializado. Na maioria das vezes, você irá querer que o servidor do notebook inicie no diretório de nível mais alto que contenha os notebooks. Geralmente este será seu diretório home. A tela deve se parecer com a tela abaixo (Fig. 1.1) Isso ainda não é o que chamamos de um notebook, mas **não entre em pânico**. O dashboard é o gerenciador de arquivos que seve para você selecionar, procurar e organizar seus notebooks. A interface é bastante simples de se entender. Vamos começar criando um diretório para guardarmos nossos projetos em Python. Para isso clique no menu `New` no canto superior direito e selecione a opção `Folder` (veja

Figura 1.1: Exemplo do Dashboard do Jupyter Notebook, a primeira tela que você deve ver quando abre o Jupyter.

a Fig. 1.2.) O diretório é criado com o nome `Untitled Folder` (i.e. dirétório sem

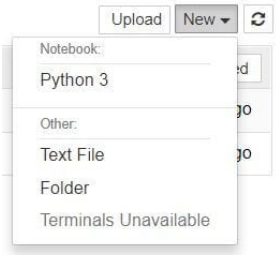

Figura 1.2: O menu `New` permite abir um novo (`Python 3`), criar um arquivo de texto (`Text File`), Folder (`Folder`) ou abrir um terminal (`Termial`).

título). Para renomear um folder ou arquivo, no dashboard, você deve marcar a caixa de seleção à esquerda do nome do folder ou arquivo e clicar na opção `Rename`, que fica disponível na barra de menu após a caixa de seleção ser ativada (veja a Fig. 1.3.)

Figura 1.3: Marcando a caixa de seleção de um arquivo ou folder você pode renomeá-lo (`Rename`), mover sua localização na árvore de diretórios (`Move`) ou deleta-lo (ícone da lixeira).

Vamos agora criar um notebook, para isso selecione a opção `Python 3` no menu `New`. Uma tela como a abaixo deverá aparecer (veja a Fig. 1.4) No menu superior,

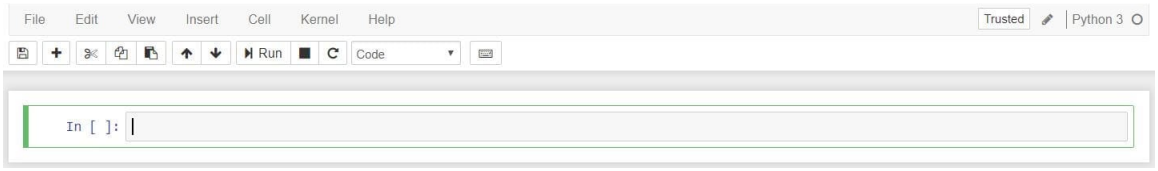

Figura 1.4: Tela inicial de um notebook

você verá opções tradicionais como `File`, `Edit`, `View`, `Insert` e `Help`, suas funções são intuitivas e você pode explora-las para conhecer suas funcionalidades. Nesse Menu aparecem dois termos importantes: `Cell` (células) e `Kernel` (núcleos).

- **Kernel:** é o nome dado pelo Jupyter Notebook a um "mecanismo computacional" que interpreta o código contido em um documento de notebook. Haverá um kernel para cada linguagem suportada pelo Jupyter Notebook. Basicamente, um kernel Python 3 conversa com o kernel do sistema operacional (no caso um kernel Linux) para executar os comandos.

- **Cell:** é um bloco para o texto a ser exibido no notebook ou código a ser executado pelo kernel do notebook.

Na Fig. 1.4, essa caixa com o contorno verde é uma célula vazia e esta marcada por `In [ ]:`. As células formam o corpo de um notebook. Existem 4 tipos de células:

1. `Code:` Células de código que contém as serem executadas no kernel. Quando o código é executado, o notebook exibe a saída abaixo da célula de código que o gerou.

2. `Markdown:` Células de texto que são formatadas usando Markdown, sua saída é exibida no local quando a célula Markdown é executada.

3. `Heading:` Células de texto formatadas como título, sua saída é exibida no local quando a célula Heading é executada.

4. `Raw NBConvert:` Células de texto que não serão formatadas.

O tipo de célula esta indicado na toolbar e nesse mesmo local podemos modificar o tipo da célula. Veja a Fig. 1.5 Os tipos mais utlizados são o `Code` e o `Markdown`. Deixemos a célula como tipo `Code`, e façamos um primeiro código simples. Comecemos

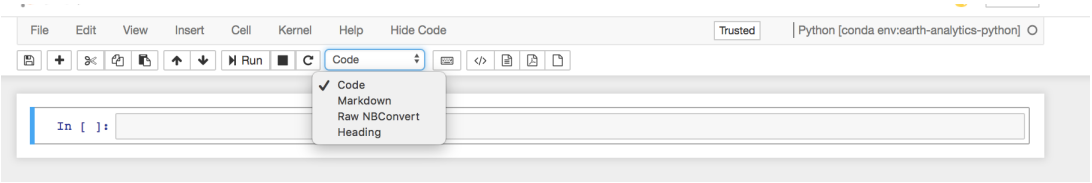

Figura 1.5: Você pode verificar o tipo de célula de qualquer célula em um Jupyter Notebook clicando na célula e olhando para o tipo de célula na toolbar.

com o clássico dos clássicos da programação digitando `print('Hello, World')`, como mostra a Fig. 1.6. A execução do código nesta célula pode ser feita clicando no ícone de execução da célula na toolbar ou pressionando as teclas `Shift + Enter`. O Jupyter

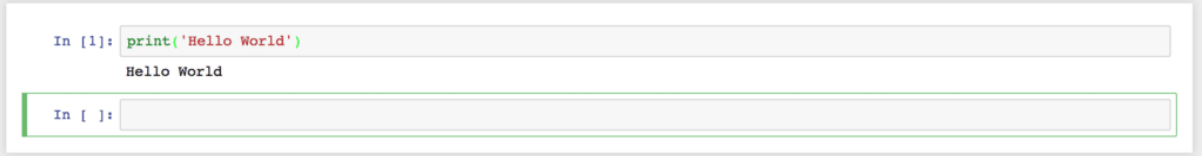

Figura 1.6: Execução da célula de código muda `In []` para `In [1]`. O comando `print` imprime o conteúdo entre parenteses, as aspas simples indicam que o conteúdo é do tipo texto. A saída da célula de código executada aparece logo abaixo da mesma.

Notebook oferece ajuda no próprio notebook para os commandos Python. Isso pode ser feito via os comandos:

1. `help()` mostra a documentação disponível para este objeto, método e função disponíveis para esse objeto.

2. `dir()` mostra as possíveis chamadas de objeto, método e função disponíveis para esse objeto.

Teste por exemplo `help(print)` e `dir(print)`. Documentações mais detalhadas para comandos mais comuns podem ser encontradas no menu `Help`. Além disso, sempre vale lembrar que existem uma farta documentação do Python e seus pacotes da internet.

O Jupyter Notebook também fornece alguns "comandos mágicos", que atalhos úties para vários problemas comuns na computação científica. Comandos mágicos vêm em dois tipos: "magias de linha", que são indicadas por um único prefixo [3] % e operam em uma única linha de entrada, e "magias de célula", que são indicadas por um prefixo

[3] caso o Jupiter Notebook esteja com a opção Automagic ligada não é necessário o % para usar "mágicas de linha".

%% e operam em toda uma célula. Um exemplo é caso em queremos saber quanto tempo levamos para executar uma instrução. Para isso podemos usar o "comando mágico" `timeit`, como no exemplo abaixo

```
%timeit list(range(100000))
```

essa instrução fará o Jupyter Notebook medir quanto tempo o sistema levará para construir uma lista de 100 mil elementos. Este comando também pode ser aplicado a toda uma célula como por exemplo:

```
%%timeit
a = list(range(100000))
b = [n + 1 for n in a]
```

aqui medimos o tempo para se criar a lista `a` e a lista `b`. Em ambos os casos o tempo deve ser da ordem de alguns milisegundos (vai variar de computador para computador). Uma lista com outros "comandos mágicos" pode ser acessada usando `%lsmagic`, referências mais detalhadas destes comandos podem ser obtidas usando `%quickref`. Veja e teste por exemplo os comandos `%hist`, `%ls`, `%magik`. Você aprenderá a usar essas ferramentas a medida que você for desenvolvendo mais e mais projetos no Python. Não se preocupe em memoriza-las.

Um grande benefício do Jupyter Notebook é que ele permite combinar código (Code) e texto (Markdown) em um documento, para que você possa documentar facilmente suas ideias e fluxo de trabalho. Isso é extremamente útil para comunicar seu trabalho a outros físicos que poderão acompanhar mais detalhadamente sua ideiam mas também é útil para te orientar onde você parou em um projeto, mantendo registrada uma ideia que você estava seguindo. Lembre-se que para deixar a célula no tipo texto selecione a opção Markdown no menu da toolbar mostrado na Fig. 1.5.

O Markdown é uma sintaxe legível por humanos (também conhecida como linguagem de marcação) para formatar documentos de texto. O Markdown pode ser usado para produzir documentos bem formatados, incluindo PDFs e páginas da web. Formatar um texto usando Markdown, é semelhante a usar as ferramentas de formatação em um editor de texto como o Microsoft Word ou o Google Docs. No entanto, em vez de usar ícones para aplicar a formatação, você usa uma sintaxe de comandos como `**essa sintaxe coloca o texto em negrito no markdown**` ou `# Aqui está um título.` A Fig. 1.7 mostra um exemplo desta formatação. Uma célula Markdown é executada da mesma forma que uma célula de código. Após executarmos

a célula com o código Markdown obtemos o texto formatado conforme as instruções. A saída do exemplo acima corresponde a Fig. 1.7 esta mostrada na Fig. 1.8.

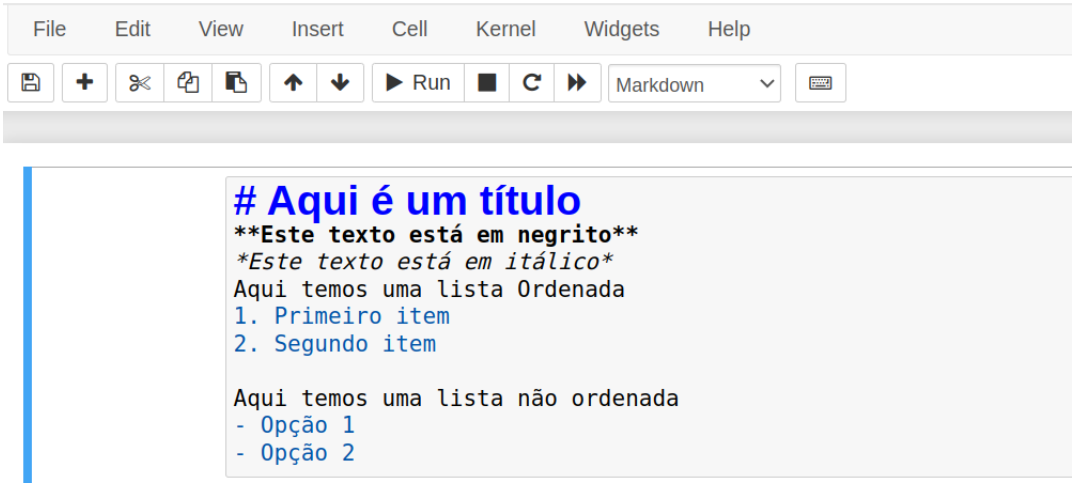

Figura 1.7: Uma célula Markdown com código antes de ser formatado. Nela vemos exemplos usuais de formatação para títulos, negrito, itálico e listas. Mais informações sobre sintaxe de formatação Markdown podem ser encontradas em markdownguide.org

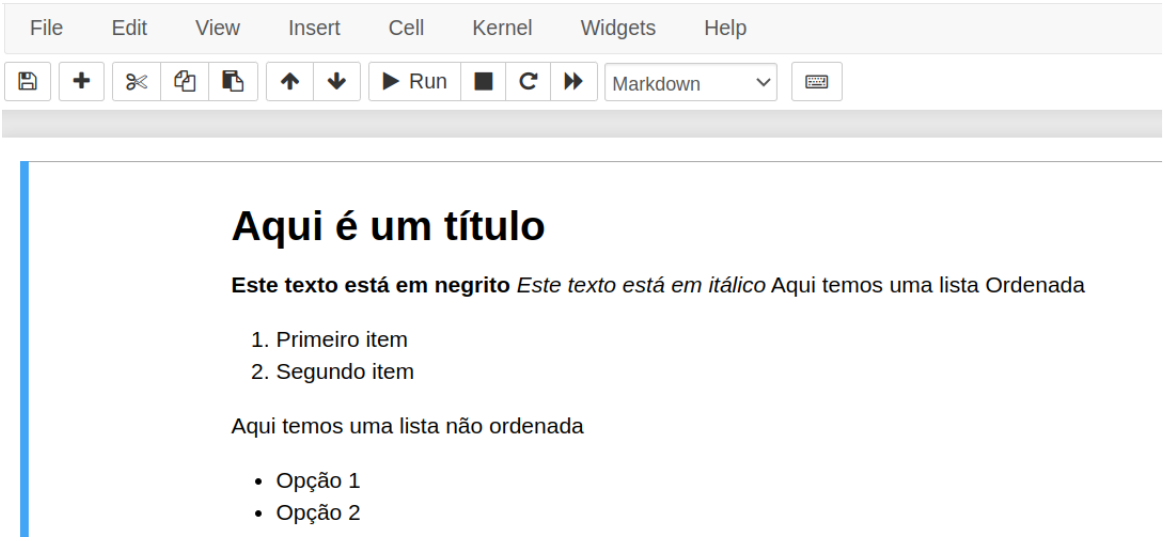

Figura 1.8: Saída de uma célula Markdown.

O Markdown também é capaz de lidar com imagens, hyperlinks e vários outras necessidades de edição de texto. Para o trabalho em física é bastante importante que possamos escrever com facilidade expressões matemáticas. Isso é feito utilizando comandos LaTeX. O LaTeXé uma linguagem de preparação de documentos para composição tipográfica de alta qualidade. É mais frequentemente usado para documentos técnicos

ou científicos de médio a grande porte, mas pode ser usado para quase qualquer forma de publicação. Este texto por exemplo foi preparado com LaTeX, assim como o são a grande maioria das publicações científicas nas áreas de física. Você pode aprender mais sobre a linguagem LaTeXno apêndice F. As Fig. 1.9 e 1.10 mostram respectivamente comandos LaTeXem uma célula Markdown e mesma célula após o processamento.

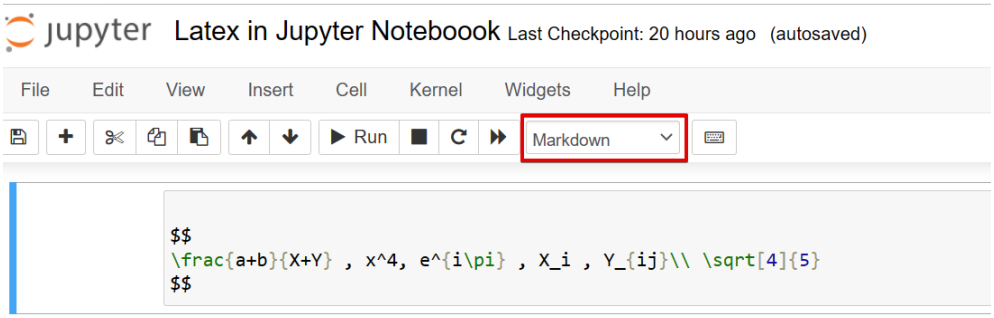

Figura 1.9: Comandos LaTeXem uma célula Markdown sendo utilizados para escrever uma equação.

$$\frac{a+b}{X+Y}, x^4, e^{i\pi}, X_i, Y_{ij}$$
$$\sqrt[4]{5}$$

Figura 1.10: Saída da célula Markdown da Figura anterior

1.3 Operações Matemáticas Simples

Até aqui, nosso conhecimento de Python nos permite fazer muito pouco, certamente nada que fosse útil para a física. A primeira coisa que precisamos conseguir utiliza-lo é entender como fazer operações aritméticas. Na maioria dos lugares onde você pode usar uma única variável em Python, você também pode use uma expressão matemática, como "x+y". Assim, por exemplo

```
x = 1
y = 2
print(x+y)
```

computador calculará a soma de x e y para você e imprima o resultado. As operações matemáticas básicas são

$x + y$	adição
$x - y$	subtração
$x * y$	multiplicação
x/y	divisão
$x * *y$	x elevado a potência$y$

Nesse ponto precisamos destacar que essas operações não agem somente em números e seu resultado depende do tipo de variáveis com as quais estamos lidando. Novamente, **não entre em pânico**, vamos discutir rapidamente sobre variáveis na próxima seção e você ter mais informações no Apêndice A.1. Mas aqui vale um exemplo simples considere o caso:

```
x = 'Fi'
y = 'si'
z = 'ca'
print(x+y+z)
n = 3
print(n*(x+y+z))
```

Nesse exemplo as variáveis x, y e z são do tipo *string* e o Python entende a operação + como concatenação dessas strings. Já n é um variável do tipo inteiro e o Python entende a operação de multiplicar um inteiro com a string x+y+z, como concatenar n vezes essa string. Essa versatilidade do Python é uma de suas características úteis, mas também pode gerar erros.

Em Python, todos os operadores matemáticos + (adição), − (subtração), ∗ (multiplicação) funcionam como seria de esperar em números e variáveis numéricas. Ou seja contanto que você fique com os tipos de variáveis numéricas, adição, subtração e multiplicação não deve haver muitos problemas. A divisão (/) tem algumas particularidades. Funciona perfeitamente em variáveis do tipo float, que são aquela usadas para descrever números via notação decimal, mas se operando entre inteiros o resultado é convertido para float. Veja os exemplos abaixo:

```
print(1/2)
print(1./2.)
print(1./2)
```

Cuidado que há uma mudança entre o Python 2 e o Python 3, neste aspecto.

O Python também possui dois outros tipos de "divisão" que são menos comuns que divisão usual, são elas:

$x//y$	O quotiente ou parte inteira da divisão
$x\%y$	O resto ou operador módulo

O quociente entre `x` e `y`, `x//y`, é a parte inteira de `x` dividida por `y`, o que significa que `x` é dividido por `y` e a parte fracionária é descartada. O operador módulo entre `x` e `y`, `x % y`, é a parte inteira restante da divisão de `x` por `y`. Por exemplo, compare:

```
print(1.5 / 0.4)
print(1.5 // 0.4)
print(1.5 % 0.4)
```

As regras de precedência em Python são exatamente o que deveriam ser em qualquer sistema matemático: (); então **; então *, /, %, // na ordem da esquerda para a direita; e finalmente +, - na ordem da esquerda para a direita.

Em Python o símbolo = não significa 'é igual a', ou seja a expressão em Python `x = y`, não é uma equação. No Python, `x = y` deve ser entendida como uma instrução para que o computador atribua a variável `x` o valor da variável `y`. O seguinte exemplo pode ser instrutivo

```
x = 1
x = x + 1
print('x =', x)
```

A primeira instrução atribui a variável `x` o valor 1, a segunda instrução (que se entendida como uma equação matemática é absurda) pega o valor atual de `x` é adiciona uma unidade. Quando a instrução `print(x)` é executada imprimisse o valor de `x` após a última instrução que o atualizou, ou seja obtemos `x = 2`. Um truque útil no Python, são as operadores *modificadores*, que permitem fazer alterações em uma variável da seguinte forma:

```
x+=y              adicione y a x, ou seja, x = x + y
x-=y              subtrair y de x, ou seja x = x - y
x*=y              multiplicar x por y, ou seja x = x*y
x/=y                divide x por y, ou seja x = x/y
x//=y  divide x por y e arredonda para um inteiro, ou seja x = x//y
```

Uma outra operação comum em Python é `x == y`, esta é uma operação de comparação e seu resultado é uma variável do tipo *booleana*, ou seja é `True` (verdadeiro) ou `False` (falso). Ou seja a instrução `x == y` é um condicional, i.e. equivale a pergunta

"o valor atual da variável `x` é igual ao valor atual da variável `y`?". Algumas outras operações de comparação (e portando com saída True ou False) comuns no Python são

`x != y`	`x é diferente de y?`
`x < y`	`x é menor que y?`
`x <= y`	`x é menor ou igual a y`
`x > y`	`x é maior que y?`
`x >= y`	`x é maior ou igual a y?`

1.4 O "Bê-a-bá" sobre variáveis.

Em programação, variáveis representam entre outras coisas instruções para computador guardar dados em sua memória, para serem usados posteriormente. São elementos fundamentais nos códigos computacionais. Por isso, vale a pena discutir como o Python lida com variáveis. Tenha em mente que o que esta escrito no lado direito do operador de atribuição = é um dado, equanto o que esta escrito no lado esquedo é um rótulo ou etiqueta para um local na memória do computador. Vamos ilustrar esse processo. Quando Python interpreta uma linha como `x = 2` , ele começa do lado direito e segue a leitura para a esquerda. Então, dada a instrução `x = 2` , o interpretador Python recebe o "2", reconhece-o como um inteiro e armazena-o em uma "caixa" de tamanho inteiro na memória. Em seguida, ele pega o rótulo `x` e usa-o como um ponteiro para esse local de memória. Esse processo é mostrado na primeira parte da Fig. 1.11.

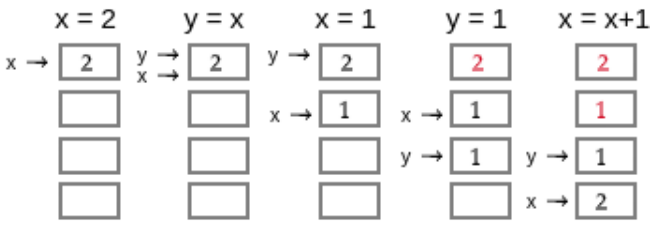

Figura 1.11: Atribuição de valores a variáveis e como o Python instruirá o computador a guardá-los na memória. Cada caixa representa uma localização na memória do computador.

Quando encontra a instrução `y = x`, o Python começa a partir da direita `x` e, reconhecendo x como um ponteiro para um local de memória, aponta o ponteiro `y` para o mesmo local de memória. Esta é a segunda parte da Fig. 1.11. Continuando nesse processo, quando receber a instrução `x = 1`, novamente o Python começa na direita e recebe o inteiro "1" e o aloca em uma 'caixa" de tamanho inteiro na memória. Só

então ele executa o lado esquerdo da instrução, isso é, desloca o ponteiro rotulado como `x` para esse novo espaço de memória. Isso não muda para onde o ponteiro `y` esta apontado. Agora quando o Python receber a instrução `y = 1`, ele ocupará um terceiro local na memória, e desviará o ponteiro `y` para este local. Este local não será o mesmo para o qual esta apontado o ponteiro `x`, para o Python não há razão para que os dois ponteiros apontem para o mesmo espaço de memória. Essa situação esta ilustrada na terceira parte da Fig. 1.11. Note que nesse momento não há nenhum ponteiro apontado para o bloco ocupado pelo valor 2. Este bloco esta ocupado com dados que não são imediatamente apagados[4], mas o sistema entende que aquele espaço de memória esta livre para ser usado.

Essa interpretação de dados a direita e rótulos a esquerda permite que você faça algumas coisas muito úteis. Por exemplo, analisemos a instrução `x = x + 1`. A direita o computador reconhecerá `x` como um ponteiro apontando para uma valor, lê esse valor é o adiciona a 1, como no final da Fig.1.11, obtendo o valor 2. Ele então desloca o ponteiro `x` para aquela nova "caixa" de memória.

Em Python também podemos atribuir múltiplos rótulos para um dado como por exemplo com a instrução `w = x = y = z = 'Fisica'`. Neste caso, cada uma dessas variáveis acabam apontando exatamente para o mesmo ponto na memória, até que sejam usadas para outra coisa. O Python também permite que você atribua dados a mais de uma variável em uma mesma instrução. Por exemplo, `a,b = 1, 2` funciona porque Python analisa os dados a direita da igualdade primeiro e reconhece um par de inteiro, então atribui esse par ao par de rótulos à esquerda. Isso é útil por nos permite trocar os valores de duas variáveis fazendo por exemplo `x, y = y, x`. Analise por exemplo o código abaixo

```
a = b = 1
a , b = b, a+b
print('a = ', a, 'b=', b)
```

Para a maior parte das aplicações, conhecer detalhadamente como o Python gerencia variáveis não é necessário.

[4] É um "lixo" que fica na memória e pode comprometer o desempenho do sistema.

1.4.1 Nomeando Variáveis

O nome de uma variável podem conter letras, números e o caractere `_`. Esses nomes nunca podem começar com um número. E é importante lembrar que o Python diferencia maiúsculas e minúsculas. Ou seja portanto, `Raio_1` não é o mesmo que `raio_1`. É uma boa prática escolher nomes de variáveis que esclareçam a função daquela variável. Por exemplo, o nome `m` é perfeitamente legal, mas quando um colega for ler o seu código o nome `massa_partícula` é muito mais descritivo. O tempo extra que você gasta digitando esses nomes mais descritivos será mais do que compensado no momento em que você economiza depurando seu código!

As variáveis também não podem receber o mesmo nome de algumas palavras reservadas tais como:

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

Tabela 1.1: Lista de Palavras Reservadas no Python 3

É possível nomear uma variável usando o nome de uma função do Python (e.g. `print`), mas a partir disso a função não ficará mais disponível. Portanto essa prática não é recomendável. Você também deve evitar nomes como "I", "l", "O" e "o" para evitar confusão com os números 1 e 0. Nomes como `n,i,j,k` são usualmente reservados para contadores inteiros. Essas e muitas outras regras e convenções estão codificadas em um guia de estilo chamado PEP8 que faz parte da documentação do Python, e pode ser acessado em https://peps.python.org/pep-0008/.

1.4.2 Alguns tipos de variáveis

Todas as variáveis em Python possuem uma certa natureza ou como se diz são de certos tipos. Isso significa que os dados que elas rotulam têm certas propriedades que ditam como elas são usadas. Tipos diferentes têm propriedades diferentes e são utilizados para diferentes aplicações. Você sempre pode determinar o tipo de uma variável (ou de um dado) usando a função interna `type()`. Veja o exemplo abaixo:

```
dimensao = 3
FortyTwo = 'Vida, Universo e Tudo Mais'
pi = 3.1415
z = 1 + 1j
Fibonacci_5 = [1, 1, 2, 3, 5]
type(dimensao),type(FortyTwo),type(pi),type(z),type(Fibonacci_5)
```

Existem diversos tipos diferentes de variáveis em Python. Os dois tipos mais comuns podem ser divididos em *tipos numéricos* e *tipos sequenciais*. Tipos numéricos contêm dados que representam um números únicos, como por exemplo "42", "3,1415" e "1 + 3i11. Tipos sequenciais contêm dados com vários objetos, que podem ser números ou caracteres, ou mesmo coleções de diferentes tipos de objetos. Um dos pontos fortes do Python é que ele automaticamente converte entre tipos de variáveis conforme é necessário e se possível. Essa mutabilidade das variáveis também pode levar a armadilhas que produzem erros no código. Discutiremos aqui os tipos numéricos e sequenciais, você pode ver mais sobre outros tipos no apêndice A.1.

Tipos Numéricos

As variáveis numéricas em Python, podem ser de 3 tipos: inteiros (type: int), números de ponto flutuante (type: float) e números complexos (type: complex).

Integer: O inteiro é o tipo numérico mais simples em Python. Os inteiros são usados para contar itens ou acompanhar quantas vezes você fez alguma coisa. Eles podem ser negativos ou positivos, ou seja eles correspondem ao que chamamos de números inteiros. Lembre que como já discutimos na seção 1.3, variáveis do tipo `int` não se dividem como esperado: Em Python, `1/2` resulta em `0`, porque 2 cabe em 1 zero vezes.

Float: O tipo "floating point" (i.e. ponto flutuante) é um número que contém um ponto decimal dividindo sua parte inteira de sua parte fracionária. Por exemplo, 3.1415 e 9.81 são do tipo float. Tipos float também podem ser escritos usando notação científica por exemplo `6.022e23` corresponde a 6.022×10^23 é um float.

Complex: Números complexos são embutidos no Python, que usa $j = \sqrt{-1}$ para unidade imaginária. Essas variáveis são do tipo $a + bj$, onde a parte real do número é

a e a parte imaginária é b. O Python executa corretamente as operações elementares com números complexos.

Números em Python N são objetos (na verdade, tudo em Python é um objeto) e têm certos *atributos*, acessados usando a notação: `<objeto>.<atributo>` (este uso do período não tem nada a ver com o ponto decimal que aparece em um tipo float. Alguns atributos são valores simples: por exemplo, objetos de números complexos têm a atributos real e imag , que são as partes real e imaginária (ponto flutuante) do número. Teste por exemplo

```
(1 + 2j).real
(1 + 2j).imag
```

Nesses objetos também é possível aplicar neles *métodos*, ou seja aplicar algumas funções no mesmo. Por exemplo

```
(1 + 2j).conjugate()
42.bit_length
(3.14159265359).bit_length
```

Digitando-se `<objeto>.` e pressionando a telca Tab o Jupyter Notebook abrirá um menu com uma lista de métodos que são possíveis de se aplicar naquele objeto.

Tipos Sequenciais

Tipos sequenciais são coleções de itens que são rotulados por um nome de variável. Os itens individuais dentro da coleção são separados por vírgulas, e referido por um índice entre colchetes após o nome da variável. Veja o exemplo:

```
Disciplinas = ('Termodinamica', 'Eletromagnetismo',
        'Quantica', 'Estado Solido')
print(Disciplinas[2])
```

O resultado destas instruções deve ser `Quantica`. Note que o índice começa a contar de 0. Você pode usar também índices negativos para contar de trás para a frente, por exemplo

```
print(Disciplinas[-1], Disciplinas[-3])
```

Vamos examinar agora alguns tipos sequencias específicos utilizados no Python.

Tuple As Tuplas são indicadas por parênteses: (). Itens em tuplas podem ser qualquer outro tipo de dados, incluindo outras tuplas. Tuplas são *imutáveis*, o que significa que

uma vez definidos seus conteúdos não podem ser alterados. Por exemplo, as instruções abaixo irão gerar um erro:

```
teste = (1,2,3)
type(a)
print(a[1])
a[1] = 'dois'
print(a)
```

List As listas são indicadas por colchetes: [ ]. As listas são praticamente o iguais às tuplas, mas são *mutáveis*: itens individuais em uma lista podem ser modificados. As listas podem conter qualquer outro tipo de dados, incluindo outras listas. Note a diferença trocando os parênteses por colchetes na primeira linha das instruções acima

```
teste = [1,2,3]
type(a)
print(a[1])
a[1] = 'dois'
print(a)
```

String Uma string é uma sequência de caracteres. As strings são delimitadas por aspas simples ou duplas: ou ' '. Strings são *imutáveis*, como tuplas. Ao contrário das listas ou tuplas, as strings só podem incluir caracteres. O exemplo abaixo mostra algumas operações com strings.

```
Palavra='trabalho e Energia'
print(Palavra)
print(Palavra[1])
print(Palavra[:5])
print(Palavra[-4:])
print(Palavra.capitalize())
print(Palavra.count('a'))
print(Palavra.split())
```

Dictionary Os dicionários são indicados por colchetes: . Eles são dif- diferente dos outros tipos sequenciais em Python, pois em vez de índices numéricos que eles usam keywords são rótulos de strings.

Range Um tipo range representa uma sequência *imutável* de números e é comumente usado para fazer um loop de um número específico de vezes em loops. Um range é

criado por uma instrução `range(start, stop, step)`, onde *start* é o índice inicial[5], *stop* o índice final (que não é incluído), e *step*[6] o intervalo entre os índices. Veja o exemplo abaixo

```
Pares = range(0,100,2)
print('Pares = ', type(Pares))
print(Pares)
print('Os primeiros pares:', Pares[0], Pares[1], Pares[2])
Pares_list = list(Pares)
print(Pares_list)
```

A última instrução mostra como converter uma variável tipo range em uma variável tipo list.

Conforme mencionado acima na descrição das listas, uma lista pode conter como elementos outras listas. Uma lista de listas lembra bastante uma matriz bidimensional. Veja por exemplo:

```
matrix = [[1,2,3],[4,5,6]]
print(matrix[1][1])
```

Que resulta em 5 (não se esqueça que os índices começam em 0). Há alguma utilidade em visualizar listas desse tipo como matrizes, mas estes objetos não operam como matrizes. Teste por exemplo `print(matrix + matrix)` e você verá que o Python apenas concatena as listas. Novamente, **não entre em pânico**, algumas bibliotecas como o SciPy e o NumPy vão adicionar funcionalidades que nos permitirão usar matrizes como conhecemos.

Listas são objetos bastante úteis e comuns. Veja o exemplo abaixo para algumas operações úteis com listas.

```
primes = [2, 3, 5, 7, 11, 13]
primes.append(17)
print(primes)
primes.reverse()
print(primes)
primes.pop(6)
print(primes)
primes.sort()
print(primes)
primes = primes+2*['a']+['b']+['c']+['b']
print(primes)
```

[5] O argumento *start* é opcional, se omitido o range começa por padrão em 0.
[6] O argumento *step* é opcional, se não incluído o intervalo padrão é 1.

```
primes.remove('a')
print(primes)
primes.pop(-1)
print(primes)
```

A tabela abaixo mostra os métodos disponíveis para operar em listas:

`append(`*element*`)`	Anexa *element* ao final da lista
`extend(`*other_list*`)`	Estende a lista com os elementos de *other_list*
`index(`*element*`)`	Retorna o índice mais baixo da lista que contém *element*
`insert(`*index, element*`)`	Insere *element* no índice *index*
`pop(`*index*`)`	Remove o elemento de índice *index*+1
`reverse()`	Reverte a ordem da lista
`remove(`*element*`)`	Remove a primeira ocorrência de *element*
`sort()`	Ordena a list
`count(`*element*`)`	Conta o número de ocorrências de *element*

Tabela 1.2: Alguns métodos disponíveis para operar em listas.

1.5 Instruções de Controle

Poucos programas de computador são executados de forma puramente sequencial, uma instrução após outra na sequência escrito no código-fonte. É comum que durante a execução do programa, dados sejam inspecionados e blocos de código executados a depender de algum teste realizado nestes dados. Instruções de controle são instruções que permitem que um programa siga caminhos diferentes dependendo de algum evento. Essas instruções fazem o papel análogo a instruções da vida real do tipo "Se você está com fome, coma." ou ainda "Enquanto a luz estiver vermelha, não entre." Todas essas instruções têm a mesma estrutura básica: (a) a própria instrução: "se" ou "Enquanto", (b) um "condicional", que é uma afirmação que deve ser avaliada como verdadeira (True) ou falsa (True): "você está com fome" ou "A luz está vermelha". E há a ação a ser executa: "coma" ou "não entre".

Nessa seção, discutiremos 3 instruções de controle do Python, são elas:

`if ... elif ... else`, `while` e `for`.

Antes de explorarmos como essas instruções funcionam vamos entender um pouco mais sobre condicionais.

1.5.1 Condicionais

Um condicional é qualquer coisa que o Python possa avaliadar como True (verdadeiro) ou falso. Geralmente um condicional será uma expressão de uma relação entre dados que são comparados usando operadores de comparação como `==, !=, <, <=, >, >=` . Veja os seguintes exemplos:

```
print(1<2)
print('a' in 'Fisica')
print('b' not in 'Fisica')
print(22/7 < 3.14)
print(10**2 == 100.)
print([1,2,3] != [1,2,'tres'])
```

Condicionais podem também ser combinados usando operadores booleanos `and, or` e `not`. Operadores booleanos tem a menor ordem de precedência em todo o Python. A tabela abaixo mostra como esses operadores booleanos agem

```
 True  and  True   -> True          True  or  True   -> True
 True  and  False  -> False         True  or  False  -> True
False  and  True   -> False        False  or  True   -> True
False  and  False  -> False        False  or  False  -> False
  not  True   -> False
  not  False  -> True
```

Tabela 1.3: Algebra Booleana

1.5.2 if ... elif ... else

A instrução de controle mais básica em Python[7], é a instrução equivalente ao "se". Ele permite que você diga ao computador o que fazer se alguma condição é satisfeita. A sintaxe é a seguinte:

```
if <condicional 1>:
    <instrucao 1>
    <instrucao 2>
    ...
elif <condicional 2>:
    <instrucao 3>
    <instrucao 4>
    ...
```

[7] ou qualquer outra linguagem de computador

```
else:
    <instrucao 5>
```

Observe a indentação do código, i.e. os espaços em branco (um Tab) entre `if ... elif ... else`. Em Python, esses espaços em branco definem que o grupo de comandos deve ser executado caso o condicional seja True. Ou seja a indentação não é opcional. A identação serve para informar ao Python, qual bloco de instruções é deve ser executado se aquele condicional for verdadeiro. As instruções `elif` (ou se) e `else` ("ou então") estendem a instrução `if`. A instrução `elif`, é uma abreviação para "else if" adiciona outro `if` que é testado se e somente se o primeiro condicional for falso. Você pode utilizar quantos `elif` forem necessários, os condicionais serão testados na ordem em que aparecerem. A instrução `else`, é executada se nenhuma das instruções `elif` anteriores, ou a inicial `if`, for verdadeira.

Exemplo: No calendário gregoriano, um ano é bissexto se for divisível por 4 com a exceção de que os anos divisíveis por 100 não são bissextos, a menos que também sejam divisíveis por 400. O programa Python a seguir determina se ano é bissexto.

Listing 1.1: Determinando se um ano é bissexto.

```
ano = int(input('Qual o ano?')

if not ano % 400:
    is_leap_year = True
elif not year % 100:
    is_leap_year = False
elif not year % 4:
    is_leap_year = True
else:
    is_leap_year = False
print('O ano ', year, ' e bissexto')
```

1.5.3 while

A instrução while é usada para repetir um bloco de comandos até que uma condição deixe de ser satisfeita. Considere o exemplo abaixo:

```
i = 0
while i < 10:
    i += 1
    print(i, end=',')
print('fim')
```

Cada vez que o bloco de instruções do `while` é executado, o contador i (que começa em 0), é acrescido de 1 e seu valor é impresso. Esse bloco será executado até que i tenha o valor 10, quando o Python sai do loop e executa o comando `print(fim)`.

Podemos combinar loops `while` com `if ... elif...else` para obtermos decisões mais complexas. Veja o exemplo a seguir:

Exemplo: Em 1937, o matemático alemão Lothar Collatz, propôs a seguinte sequência de inteiros positivos: O primeiro termos da sequência é $c_1 = n$, onde n é um número inteiro positivo. Dado o c_k, o k-ésimo termo da sequência, o termo seguinte é obtido fazendo

$$a_{k+1} = \begin{cases} a_k/2 & \text{se } a_k \text{ é par} \\ 3a_k + 1 & \text{se } a_k \text{ é ímpar} \end{cases}$$

Collatz conjecturou que qualquer que seja o inteiro inicial c_1 a sequência sempre termina em 1. Até o momento em que escrevo essas notas (2022), a Conjectura de Collatz nunca havia sido provada, permanecendo como um problema em aberto (para mais detalhes veja a Ref. [8]. Nesse exemplo, iremos obter a sequência de Collatz para um inteiro positivo qualquer. Tente resolver antes de checar uma possível solução abaixo.

Listing 1.2: Testando a conjectura de Collatz

```
num = int(input('Qual o inteiro positivo inicial?'))
print('A sequencia de Collatz para', num)
while num != 1:
    if num % 2 == 0:
        num = int(num/2)
        print(num, end=',')
    else:
        num = int(3*num+1)
        print(num, end=',')
```

Loops `while` podem ser estendidos usando alguma palavras chave:

continue A instrução `continue` move a execução do programa para o topo o bloco `while` sem terminar a parte do bloco seguinte a instrução continue.

break A instrução `break` interrompe a execução do loop e recomeça o código diretamente para a linha seguinte ao bloco `while`. Em outras palavras, ele "sai" do loop.

else Um comando `else` no final de um bloco `while` é usado para delinear um bloco de código que é executado depois que o bloco `while` termina, mas o código neste bloco não é executado se o bloco `while` for encerrado via um comando de pausa.

O exemplo a seguir deve esclarecer o uso dessas instruções.

Exemplo: Você precisa escrever um programa que teste se um número é primo ou não. O programa deve pedir o inteiro para testar, em seguida, imprima uma mensagem informando o primeiro fator encontrado ou informando que o número é primo. Tente você mesmo antes de examinar o exemplo abaixo:

Listing 1.3: Testando se um número é primo

```
Number = int(input('Qual inteiro iremos testar?'))
TestNumber = 2
while TestNumber < Number :
    if Number % TestNumber == 0 :
        print(Number , 'e divisivel por' , TestNumber , end='.')
        break
    else:
        TestNumber += 1
else:
    print(Number , 'e primo', end ='.')
```

Esta não é a forma mais eficiente de determinar se um número inteiro muito grande é primo.

1.5.4 for

O loop `for` itera sobre os itens em uma sequência, repetindo o bloco de loop uma vez por item. A sintaxe mais básica é a seguinte:

```
for <iterador> in <tipo sequencial>
    <instrucao 1>
    <instrucao 2>
    ...
```

Cada vez que passar pelo loop, o valor do `iterador` será o valor do próximo elemento em `<tipo sequencial>` . Vejamos o exemplo abaixo,

```
lista = [1,'a',[1,2]]
index = 0
for t in lista:
    print('o elemento', index, '=', t)
    index += 1
```

o código imprime cada um dos elementos de `lista`. Você pode usar qualquer tipo sequencial, mas tipos imutáveis são preferíveis para se evitar alterar a sequência que esta controlando o loop. Por exemplo, o exemplo abaixo vai gerar um loop infinito:

```
lista = [1,'a',[1,2]]
```

```
index = 0
for t in lista:
    print('o elemento', index, '=', t)
    lista += index*[t]
    index += 1
```

O exemplo abaixo mostra como podemos usar uma string para controlar o `for`.

```
letras = 0
for t in 'Palavra':
    letras +=1
```

No cálculos numéricos, é mais comum usar o comando for sobre um intervalo numérico gerado com `range`

```
N, S = 100, 0
for t in range(0,N,1):
    S += t
print(S)
```

O exemplo abaixo mostra como gerar a sequência de Fibonacci e aproximar a razão áurea.

Listing 1.4: Sequência de Fibonacci e a Razão Áurea

```
N = 100
fibonacci = [1, 1]

type(fibonacci)
for n in range(2,N):
    fibonacci.append(fibonacci[n-1]+fibonacci[n-2])
print(fibonacci)
print('Razao Aurea aproximadamente', fibonacci[n]/fibonacci[n-1])
```

A instrução `for` também permite o uso das mesmas palavras-chave usadas para `while`. Além, disso o Python fornece uma sintaxe muito útil e concisa para criar e manipular tipos sequenciais. Quando aplicada a listas essa sintaxe é chamada de *List Comprehensions*. A ideia é usar instruções `for` e `if` no interior da indicação de uma lista. A sintaxe é

```
lista_1 = [ <expressao> for <iterador> in <tipo sequencial]
lista_2 = [<expressao> if <condicional> else <expressao>
        for <iterador> in <tipo sequencial>]
```

O uso de *List Comprehensions* deve ficar mais claro nos exemplos abaixo:

Exemplo: Suponha que você tenha uma lista de medidas de comprimento feitas em polegadas. Precisamos converter essas medidas para centímetros (1 polegada

= 2.54 cm).

```
lista_de_medidas = [1.1, 1.3, 1.75, 2.1, 9.1, 10.2, 9.8 ]
medidas_cm = [2.54*medida for medida in lista_de_medidas]
```

Exemplo: Crie uma lista com os números entre 0 e 100 que são divisíveis por 6, mas não são divisíveis por 5.

```
max = 100//6
numeros=[6*n for n in range(max) if (not 6*n % 5 ==0)]
print(numeros)
```

1.6 Funções

Uma função é um bloco de código que recebe seu próprio nome para que possa ser usado repetidamente por várias partes de um programa. Nesse sentido uma função pode ser um código para realizar um cálculo matemático, ou um código para fazer algo, como desenhar um gráfico ou salvar uma lista de números. No Python, a primeira linha de uma função é definida com a palavra-chave `def` (vem de *definition*) seguido pelo nome da função, a lista de argumentos e dois pontos(:). As demais linhas contém as operações que compõem a função, que ficam em um bloco de código indentado. Lembre que no Python, a indentação funciona como um organizador para que o Python saiba a qual parte do código aquele bloco pertence. A sintaxe basica de uma função é a seguinte

```
def <nome>(<argumentos>):
    <codigo da funcao>
```

O `<nome>` estabelece o nome pelo qual a função será chamada. As mesmas regras para se nomear uma variável (ver 1.4.1, valem para nomear funções. As funções no Python, podem possuir argumentos, de forma análoga a suas análogas em matemática. Esses argumentos são uma lista de coisas que a função precisa para fazer seu trabalho. Os argumentos vem dentro de parênteses imediatamente após o nome da função. O código indentado será executado sempre que a função for chamada e usará os argumentos dados entre parênteses como parâmetros. As funções também podem retornar valores usando a palavra-chave `return` seguida por uma expressão a ser devolvida. Neste caso a syntaxe é

```
def <nome>(<argumentos>):
    <codigo da funcao>
    return <retorno1>, <retorno2>, ..., <retorno3>
```

Exemplo: Vamos escrever uma função que soma os dígitos de um número. Tente você antes de ver o exemplo abaixo.

```
def soma_digitos(n):
    n_name = str(n)
    s = 0
    for i in n_name:
        s += int(i)
    return s
```

A função também pode retornar múltiplos valores. Vejo o exemplo

```
def energia_momento(m,v):
    K = (1/2)*m*v**2
    p = m * v
    return K, p
```

Em várias situações é interessante definir argumentos com valores padrão. Em Python, eles são conhecidos como argumentos de palavras-chave, e nesse caso apontar o valor dos argumentos passa a ser opcional, i.e. a função usa o argumento padrão quando um novo valor não é usado. Quando os argumento são passados com palavras-chave, eles podem ser chamados em qualquer ordem. Os argumentos de palavra-chave são definidos pelo nome do argumento (a palavrachave), um sinal de igual (=), e o valor padrão que é usado se o argumento não for fornecido quando a função é chamada. A sintaxe básica é a seguinte

```
def <name>s(<key word1> = <val1>, ..., <key wordn> = <valn>):
    <codigo da funcao>
```

1.7 Bibliotecas: o Básico

Uma das coisas mais úteis do Python é a existência de inúmeras bibliotecas que permitem adicionar novas funcionalidades ao sistema. Sem elas o Python básico é apenas um linguagem de programação e teríamos que escrever códigos até para coisas bastante simples. Por exemplo, sozinho o Python não possui funções trigonométricas embutidas, nem conhece o valores de π e e. Mas isso pode ser facilmente resolvido importando a biblioteca **math**. Isso é feito com o comando `import math`

Os comandos `import`são geralmente colocados no início do código, mas essa é meramente uma questão de conveniência. Uma vez que o pacote/biblioteca tiver sido

importado, todas as funções no **math** estão disponíveis e podem ser acessadas usando a sintaxe `math.<function-name>`. Por exemplo

```
import math
x = math.pi/3.0
print(x, math.sin(x), math.cos(x), math.tan(x))
```

Existem muitas outras funções e constantes no pacote **math**. Você pode saber quais usando

```
help(math)
```

Algumas vezes é útil para apenas importar elementos individuais de um pacote em vez do pacote inteiro. Por exemplo para que pudéssemos nos referir a `sin(x)` ao invés de `math.sin(x)`.

```
from math import sin
```

Outras vezes você pode importar todas as funções em um pacote (e novamente se referir a função pelo nome):

```
from math import *
```

Uma outra opção é dar um apelido curto para o pacote durante a importação. Por exemplo, durante todos esse texto utilizaremos um pacote bastante útil para a produção de gráficos chamado **matplotlib.pyplot** é conveniente usar

```
import matplotlib.pyplot as plt
```

Com isso os comandos do **matplotlib.pyplot** são chamados usando `plt.<function-name>`. Vamos usar essa oportunidade para construir nosso primeiro gráfico. Vamos plotar os pontos $(0,0),(1,2),(3,1)$. Para isso (incluindo o comando de importação do **matplotlib.pyplot**, que não será necessário caso você já tenha feito a importação)

```
import matplotlib.pyplot as plt

plt.scatter([0,1,3],[0,2,1])
plt.show()
```

Você deve obter algo como mostrado na imagem 1.12

Construção de um barra de progresso para algorítimos que levem tempo

```
from tqdm.notebook import tqdm
from time import sleep

for i in tqdm(range(10)):
    sleep(3)
```

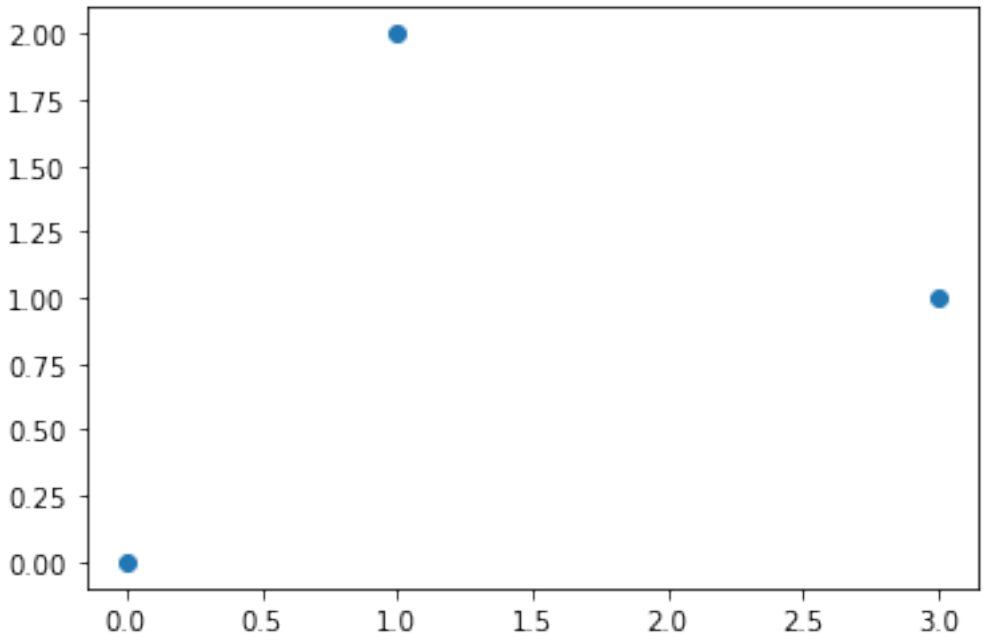

Figura 1.12: Plot dos pontos $(0, 0), (1, 2), (3, 1)$.

1.8 Gráficos Simples

1.9 Lendo dados de arquivos

1.10 Exercícios

1. Comente com # os códigos dados como exemplo nesse capítulo, indicado o que cada linha ou bloco instrui o computador a fazer.

2. Escreva um programa que pede para o usuário escrever um número inteiro entre 0 e um milhão, e então escreve esse número por extenso.

3. Escreva uma função que calcule o fatorial de um número inteiro. A seguir compare a eficiência do seu código com as funções factorial disponíveis nos pacotes **math** e **numpy**. Use o `%timeit` para isso.

4. Escreva uma função para calcular a função $\text{sinc}(x)$ que dada por

$$\text{sinc } x = \frac{\sin x}{x}.$$

   Garanta que a função calcule corretamente o caso $x = 0$ (esse é um dos limites fundamentais que você aprendeu em Cálculo I).

5. Um número T é chamado triangular quando existe um inteiro N tal que

$$T = 1 + 2 + \cdots + N.$$

Por exemplo, 3(= 1+2), 6(= 1+2+3), 10(= 1+2+3+4) são números triangulares. Construa uma função que calcula o n-ésimo número triangular.

6. Quando escrevemos um número n na base decimal estamos implicitamente dizendo:

$$n = d_N 10^N + d_{N-1} 10^{N-1} + \cdots + d_2 10^2 + d_1 10^1 + d_0 10^0$$

onde $N+1$ é o número de dígitos de n e $d_k = 0, \cdots, 9$, $k = 0, \cdots, N$ são os dígitos desse número ($d_N \neq 0$). Da mesma forma n pode ser escrito na base binária, onde os digitos dão $b_j = 0, 1$ fazendo

$$n = b_m 2^m + b_{m-1} 2^{m-1} + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

É claro que o maior número que pode ser escrito em base binária com m dígitos é

$$n_{max} = 2^m + 2^{m-1} + \cdots + 2^2 + 2^1 + 2^0 = 2^{m+1} - 1$$

resultado que obtemos reconhecendo a série geométrica finita na expressão acima. Vamos escrever um código que converte um número da base decimal para a base binária.

7. Um problema clássico em olimpíadas de matemática para o nível fundamental é o *Problema dos Armários*. O problema é o seguinte: Em uma escola existem 1000 armários numerados (de 1 a 1000) e 1000 alunos. No primeiro dia de aula, o Diretor reúne todos os 1000 alunos em fila. O Diretor então manda que o primeiro aluno da fila abra todos os armários. Após esse aluno terminar a tarefa, o diretor ordena que o próximo estudante (i.e. o segundo aluno da fila) feche todos os armários com número par. Quando ele conclui a tarefa o diretor, manda que o terceiro aluno da fila mude o estado de todos os armários com número divisível por 3. Mudar o estado significa se o armário estiver aberto, ele irá fecha-lo e se o armário estiver fechado, ele irá abri-lo. Ao quarto aluno, o diretor mandará que mude o estado de todos os armários divisíveis por 4, ao quinto aluno que mude o estado de todos os armários divisíveis por 5, e assim sucessivamente até que todos os 1000 alunos tenham participado.

(a) Escreva um programa para simular o problema descrito. O programa deve ser capaz de determinar aos final do processo: quais armários estão abertos, quais estão fechados e o número total de armários abertos.

(b) Como disse na introdução este é um problema de matemática. A resolução esperada do *Problema dos Armários* não envolve computação, mas sim a compreensão de propriedades de divisores de números inteiros. Vamos tentar entender essa propriedade usando um programa de computador: Construa um programa que analisa os 10 mil primeiros números inteiros e separa em uma lista aqueles que tem um número ímpar de divisores. Verifique que todos estes números são quadrados perfeitos. Suponha que esta é uma propriedade geral, como ela pode ser usada para responder o *Problema dos Armários*.

(c) **(Opcional)** Demonstre matemáticamente a propriedade conjecturada no item anterior, i.e. "Um núemero inteiro positivo é um quadrado perfeito se e somente se ele possui um número ímpar de divisores".

Capítulo 2

Cinemática

Problemas do Haliday Capítulos 2 a 4

Problema 1: Uma bola é lançada de uma torre de altura h com velocidade inicial zero. Escreva um programa que pede ao usuário para digitar a altura em metros da torre e depois calcula e imprime o tempo que a bola leva até atingir o solo, ignorando a resistência do ar. Use seu programa para calcular o tempo para uma bola cair de uma torre de 100 m de altura

Capítulo 3

Dinâmica

Problemas do Haliday Capítulos 5 e 6

Problem 2 The drag force on a skydiver is of the form

$$F_a = -kv^2$$

where $k = 0.7kgm^{-1}s^2$ with the parachute closed and $k = 30kgm^{-1}s^2$ with the parachute open. The skydiver performs a series of 3 jumps from an altitude of $3000m$. To perform the jump safely, the speed of the skydiver must be less than 10m/s when they land.

Use the scipy.integrate.odeint function to solve the differential equation describing the sky-diver's motion. See the Session 4 Advanced Computing Worksheet for help using this function. For each jump listed below plot the altitude and velocity of the skydiver against time and calculate the total time taken for the jump.

a) The parachute is open for the whole jump

b) The jump is performed using a static line 1000 m in length. Therefore the parachute opens when the skydiver is 1000 m below the plane

c) The skydiver deploys their parachute to minimise the total time taken for the jump, whilst still landing safely

Capítulo 4

Trabalho e Energia

This question concerns the properties of the potential function found in the file Potential.txt.

a) Plot the potential given by the data

b) Find the equilibrium point(s) of the potential and show if they are stable or unstable

Note

You can use scipy.interpolate.interp1d to create a function which you can then solve using fsolve. Use the xtol parameter to specify a sensible value for the tolerance of the solution found by fsolve.

c) Find and plot the maximum energy of particle bound in this potential

d) Find and plot the allowed region for this bound particle

Listing 4.1: Plot dos dados disponíveis no arquivo Potential.txt

```
import numpy as np
from matplotlib import pyplot as plt

x, y = np.loadtxt("Potential.txt", unpack=True)

plt.grid()
plt.plot(x, y)
plt.xlabel("x")
plt.ylabel("V(x)")
plt.show()
```

Listing 4.2: Análise dos derivadas do potencial dados disponíveis no arquivo Potential.txt

```
# Import scipy interpolation
from scipy.interpolate import interp1d
from scipy.optimize import fsolve
```

```
# Find first and second derivatives of the potential
derivative = np.gradient(y, x)
second_derivative = np.gradient(derivative, x)

# Create a function using scipy.interpolate.interp1d
f_derivative = interp1d(x, derivative)  # interp1d returns a function given a set of x c

# Use fsolve to find zero points
zero_point1 = fsolve(f_derivative, 1)
zero_point2 = fsolve(f_derivative, 2.7, xtol=1e-5)

# Interpolate to find the value of the potential and the second derivative at the turnin
f_potential = interp1d(x, y)
f_second = interp1d(x, second_derivative)

zeros = [zero_point1, zero_point2]

for i, zero in enumerate(zeros):
    # Find values of potential and second derivative
    zero_y = f_potential(zero)
    zero_point_second = f_second(zero)

    # Print and plot zero points to 3 s.f.
    print("Zero point %i is as x = %.3f" % (i + 1, zero))
    print("The y value at this point is %.3f" % zero_y)

    # Work out if equilibrium is stable
    if zero_point_second < 0:
        print("This is an unstable equilibrium point")
    elif zero_point_second > 0:
        print("This is a stable equilibrium point")
    else:
        print("This equilibrium point is indeterminate. Examine the graph or higher d

    # Plot zero points
    plt.plot(zero, zero_y, "o", color="b")

plt.plot(x, y, label="Potential")
plt.plot(x, derivative, label="1st Derivative")
plt.plot(x, second_derivative, label="2nd Derivative")
plt.grid()
plt.xlabel("x")
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))  # Put legend outside plot
plt.show()
```

Find the maximum energy of a bound particle The maximum energy of the bound particle is just less than value of the unstable equilibrium point

Listing 4.3: Energia Máxima de uma partícula presa pelo potencial. Dados disponíveis no arquivo Potential.txt

```
E_max = f_potential(zero_point2)
print("The maximum allowed energy of the particle is %.3f" % E_max)
```

```
plt.plot(x, y, label="U(x)")
plt.axhline(E_max, linestyle="--", label="Maximum bound energy", color="k")
plt.grid()
plt.xlabel("x")
plt.legend()
plt.show()
```

Listing 4.4: Região permitida para a partícula presa pelo potencial. Dados disponíveis no arquivo Potential.txt

```
# Solve for the point where the lines U(x) and E_max intersect
# i.e. U(x) - E_max = 0

def f_intersection(x_in):
    intersect = f_potential(x_in) - E_max
    return intersect

x1 = fsolve(f_intersection, 0.5, xtol=1e-5)
x2 = zero_point2

print("The particle is bound between the points %.3f and %.3f" % (x1, x2))

plt.plot(x, y, label="U(x)")
plt.axhline(E_max, linestyle="--", label="Maximum bound energy", color="k")
plt.axvline(x1, linestyle="--", color="k")
plt.axvline(x2, linestyle="--", color="k")
plt.plot([x1, x2], [E_max, E_max], "o", color="b")
plt.grid()
plt.xlabel("x")
plt.legend()
plt.show()
```

Efeito resistivo do ar em bicicletas lançamento de projéteis ver https://drive.google.com/fil
S1tEZ88sp/view

Capítulo 5

Momento Linear

Problemas do Haliday Capítulos 9

Listing 5.1: Determinando Centro de Massa de uma distribuição aleatória em 2D

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n =100
# Gerar posicoes e Massas aleatorias
x = np.random.randint(-50, 50, n)
y = np.random.randint(0,200,n)
m = np.random.randint(1,200, n)

# Calcula as Coordenadas do Centro de Massa
cgx = np.sum(x*m)/np.sum(m)
cgy = np.sum(y*m)/np.sum(m)

# Plot do Centro de Massa
plt.scatter(x,y,s=m);
plt.scatter(cgx, cgy, color='k', marker='+', s=1e4);
plt.title('Centro de Massa');
```

Capítulo 6

Rotações, Torque e Momento Angular

Problemas do Haliday Capítulos 10 e 11

Capítulo 7

Gravitação

Problema 1 Um satélite deve ser lançado em uma órbita circular ao redor da Terra de modo que orbite o planeta uma vez a cada T segundos. (a) Mostre que a altitude h acima da superfície da Terra que o satélite deve ter é

$$h = \left(\frac{GMT^2}{4\pi^2}\right)^{\frac{1}{3}} - R$$

, onde $G = 6.67 \times 10^{-11} m^3 kg^{-1} s^{-2}$ é a constante gravitacional de Newton, $M = 5.97 \times 10^{24} kg$ é a massa da Terra e $R = 6371 km$ é o seu raio. b) Escreva um programa que peça ao usuário para inserir o valor desejado de T e então calcule e imprima a altitude correta em metros. c) Use seu programa para calcular as altitudes dos satélites que orbitam a Terra uma vez por dia (a chamada órbita "geossíncrona"), uma vez a cada 90 minutos e uma vez a cada 45 minutos. O que você conclui do último desses cálculos? d) Tecnicamente um satélite geossíncrono é aquele que orbita a Terra uma vez por dia sideral dia, que é 23,93 horas, não 24 horas. Por que é isso? E quanta diferença fará à altitude do satélite?

Listing 7.1: Altitude de um Satélite

```
from math import pi

# parametros e constantes kg, m, s
G = 6.67*10**(-11)
M = 5.97*10**(24)
R = 6371*10**3

#input
unit=input('Qual a unidade utilizada (dias, horas, minutos, segundos)?')
period = float(input('Qual o periodo?'))
if not unit in ['dias', 'horas', 'minutos', 'segundos']:
    print('Unidade nao Reconhecida')
elif unit == 'dias':
```

```
    period = 24*60*60*period
elif unit == 'horas':
    period = 60*60*period
elif unit == 'minutos':
    period = 60*period

h = (G*M*period**2/(4*pi**2))**(1/3)-R
print(h, 'm')
```

Problemas do Haliday Capítulo 13

1. Larry Niven wrote a series of science fiction books about Ringworld, an inhabited, manufactured ring of metal that cir- cled a star. Consider a uniform ring of material with total mass M and radius R. Assume that the ring is infinitesimally thin. In terms of G, M, and R, (a) calculate the gravitational potential energy at a point r = R /2 in the plane of the ring, and (b) calculate the magnitude and direction of the force of gravity on a 1-kg mass located at that same point. (c) Repeat (a) and (b) for a point r = 3R /2 in the plane of the ring. (See "Bound Orbits with Positive Energy," by J. West, S. Das- sanayake, and A. Daniel, American Journal of Physics, Janu- ary 1998, p. 25.) 2. Repeat Problem 23 for 19 particles, 29 particles, 39 particles, and so on up to 99 particles. Plot the results on a graph of number of particles versus rotational period. Does the result converge to a limit as the number of particles becomes infi- nite? If so, what is that limit? Can the problem be solved ana- lytically?

Problema de Kepler 2 corpos

Ainda possui um erro

Listing 7.2: Orbitas: Problema de Kepler para 2 Corpos

```
import numpy as np
import matplotlib.pyplot as plt

#Parameters
GM = 1
pi = np.pi

#Simulation Parameters
dt = .001 # time step
T  = 10 # total simulation time, T/dt will give the total number of steps

# Deal with polar coordinates
def polar_coord(x, y):
    r = np.sqrt(x**2+y**2)
    if x == 0:
        theta = np.sign(y)*pi/2
    else:
        theta = np.arctan(y/x)
```

```
    return r, theta

def polar_vector(x,y,vx,vy):
    r, theta = polar_coord(x,y)
    hatr = np.array([np.cos(theta), np.sin(theta)])
    hattheta = np.array([-np.sin(theta), np.cos(theta)])
    vpolar = np.linalg.solve([hatr, hattheta], [vx, vy])
    vr, vtheta = vpolar[0], vpolar[1]
    return vr, vtheta

class Planet: # Create Class Planet
    def __init__(self, x, y, vx, vy, mass):
        self.mass = mass
        r0 , theta0 = polar_coord(x,y)
        self.r, self.theta = r0, theta0
        self.rtraj = [self.r]
        self.thetatraj = [self.theta]
        self.vr, self.vtheta = polar_vector(x, y, vx, vy)
        self.L = mass*r0**2*self.vtheta
        self.energy = (1/2)*mass*(vx**2+vy**2)-GM*mass/r0

    def move(self): # Move Planet
        self.r += dt*self.vr #Up date r positions
        self.theta += dt*self.vtheta #Up date Y positions
        self.rtraj = np.append(self.rtraj, self.r) #Up date r trajectory
        self.thetatraj = np.append(self.thetatraj, self.theta) #Up date theta traje
        potential_f = - GM*self.mass/self.r**2
        centrifugal_f = self.L**2/(self.mass*self.r**3)
        f_eff = potential_f +  centrifugal_f
        self.vr +=  dt*f_eff/self.mass
        self.vtheta = self.L/(self.mass*self.r**2)

# Create a planet
planet = Planet(1,0,0.1,1,1)
t = 0
while planet.theta <= 2*pi:
    planet.move()
    t += dt
    if t >= 10**9:
        break
    else:
        continue

r_max = np.amax(planet.rtraj)
r_min = np.amin(planet.rtraj)
print('Periodo orbital =', t)
print('Afelio =', r_max)
print('Perielio =', r_min)
print('excentricidade=', (r_max - r_min)/(r_max+r_min))

#Plot Orbit

fig, ax = plt.subplots(subplot_kw={'projection': 'polar'})
ax.plot(planet.thetatraj, planet.rtraj)
```

```
ax.plot(planet.thetatraj[0], planet.rtraj[0],'ro', markersize=8)
ax.plot(planet.thetatraj[-1], planet.rtraj[-1],'b+', markersize=8)
ax.set_rmax(1.1*r_max)
ax.grid(True)
ax.set_title("Motion of the Planet", va='bottom')
plt.show()
```

Problema de Kepler Sol-Terra-Lua ver

https://drive.google.com/file/d/1maO81Tll58t7KZXXaIZzalZ3JtpMEsio/view

Capítulo 8

O Oscilador Harmônico

Problemas do Haliday Capítulo 15

Listing 8.1: Oscilador Harmônico Simples

```
import matplotlib.pyplot as plt
import numpy as np

# set constants
k = 10
m = 1
t_max = 10.0

#Set the number of iterations to be used in the for loop
no_of_iterations=100000

# set time step so that the loop will always iterate until t=t_max seconds
dt = t_max/no_of_iterations


# make arrays to store data
t = np.zeros(no_of_iterations)
x = np.zeros(no_of_iterations)
v = np.zeros(no_of_iterations)

# set initial conditions
t[0] = 0
x[0] = 5
v[0] = 0


# evolve
for i in np.arange(1,no_of_iterations):
    t[i] = dt * i
    v[i] = v[i-1] - dt *k/m*x[i-1]
    x[i] = x[i-1] + dt * v[i]

fig, ax = plt.subplots(2, 2,figsize=(10, 10))
ax[0,0].plot(t, x)
ax[0,0].set(title = 'time x position', xlabel='time', ylabel='position')
```

```
ax[1,0].plot(t, v)
ax[0,0].set(title = 'time x velocity', xlabel='time', ylabel='velocity')
ax[0,1].plot(x,v)
ax[0,1].set(title = 'position x velocity', xlabel='position', ylabel='velocity')
ax[1,1].plot(t,1/2*m*v**2+1/2*k*x**2)
ax[1,1].set(title = 'time x energy', xlabel='time', ylabel='energy')
```

Listing 8.2: Oscilador Harmônico Duplo

```
import matplotlib.pyplot as plt
import numpy as np

# set constants
k = 10
m = 1
t_max = 10.0
l = 1
#Set the number of iterations to be used in the for loop
no_of_iterations=1000

# set time step so that the loop will always iterate until t=t_max seconds
dt = t_max/no_of_iterations


# make arrays to store data
t = np.zeros(no_of_iterations)
x1 = np.zeros(no_of_iterations)
v1 = np.zeros(no_of_iterations)
x2 = np.zeros(no_of_iterations)
v2 = np.zeros(no_of_iterations)

# set initial conditions
t[0] = 0
x1[0] = 0
v1[0] = 5
x2[0] = 0
v2[0] = -5


# evolve
for i in np.arange(1,no_of_iterations):
    t[i] = dt * i
    v1[i] = v1[i-1] - dt *k/m*(x1[i-1]-x2[i-1])
    v2[i] = v2[i-1] + dt *k/m*(x1[i-1]-x2[i-1])
    x1[i] = x1[i-1] + dt * v1[i]
    x2[i] = x2[i-1] + dt * v2[i]

fig, ax = plt.subplots(2, 2,figsize=(10, 10))
ax[0,0].plot(t, x1,label="particle 1")
ax[0,0].plot(t, x2,label="particle 2")
ax[0,0].set(title = 'time x position', xlabel='time', ylabel='position')
ax[0,0].legend(loc='upper left')
ax[1,0].plot(t, v1,label="particle 1")
ax[1,0].plot(t, v2,label="particle 2")
```

```
ax[1,0].set(title = 'time x velocity', xlabel='time', ylabel='velocity')
ax[1,0].legend(loc='upper left')
ax[0,1].plot(t,(1/2)*m*v1**2+(k/2)*x1**2)
ax[0,1].plot(t,(1/2)*m*v2**2+(k/2)*x2**2)
ax[0,1].set(title = 'time vs energy', xlabel='time', ylabel='energy')
ax[1,1].plot(t,(x1+x2)/2)
ax[1,1].set(title = 'Movement of the center of mass', xlabel='time', ylabel='Center
```

Listing 8.3: Rotores acoplados

```
import matplotlib.pyplot as plt
import numpy as np

# parameters

damping = 0.1
freq_nl = 5.0
total_time = 20
dt = 0.0001
steps = int(total_time/dt)
pi = np.pi

#arrays

theta1 = np.zeros(steps)
theta2 = np.zeros(steps)
v1 =  np.zeros(steps)
v2 =  np.zeros(steps)
t = dt*np.arange(steps)

# Initial conditions
theta1[0] = 0
theta2[0] = pi/3
v1[0] = 0
v2[0] = 0


for i in range(steps-1):
    theta1[i+1] = theta1[i] + dt*v1[i]
    theta2[i+1] = theta2[i] + dt*v2[i]
    v1[i+1] = v1[i] + dt*(freq_nl*np.sin(theta1[i+1]-theta2[i+1]) - damping*v1[i+1]
    v2[i+1] = v2[i] + dt*(freq_nl*np.sin(theta2[i+1]-theta1[i+1]) - damping*v2[i+1]

# Plots
plt.plot(t,theta1, label = "rotor 1")
plt.plot(t, theta2, label = "rotor 2")
plt.legend()
plt.show()
```

1. Consider a system composed of two objects constrained to move along the x axis. The first object is connected to a spring that is attached to the origin, and the second object is connected to a spring that is attached to the first object. Both objects

have the same mass, 0.10 kg, and both springs have the same force constant, 1.0 N/m. (a) Numerically simulate the motion of the objects, assuming that the second object is pulled and then released a distance of 1.0 cm from the equilibrium position. Generate a graph of the mo- tion of the objects. (b) Use a fast Fourier transform (avail- able on some spread sheet programs) to show that there are two characteristic frequencies of the motion. What are these frequencies?

2. An object of mass m moves subject to a force that results in a potential energy of U(x) = 14 kx 4 . This type of motion is called a quartic oscillator. Note that the frequency of oscillation de- pends on the amplitude of the oscillations here. Assuming a mass m = 0.10 kg and a force constant of k = 100 N/m 3 , nu- merically simulate the motion for several different ampli- tudes. Graph the results, and find the relation between ampli- tude and frequency for this system.

Pêndulo Simples além das pequenas oscilações ver https://drive.google.com/file/d/13ts72fG1Y2S1tEZ88sp/view

Capítulo 9

Ondas

Problemas do Haliday Capítulo 16 e 17

A simple function is given by $y(x) = x(\pi - x)$ in the region $0 \leqslant x \leqslant \pi$. It is desired that this function be approximated by a series of sine functions in the form

$$y(x) = a_1 \sin x + a_3 \sin 3x + a_5 \sin 5x \cdots$$

Esse é um exemplo de uma aproximação por série de Fourier. (a) Use a graphing program and estimate the values for a_1, a_3 , and a_5 that give the best visual fit. (b) Use a symbolic math program (such as Scipy or Scilab) to evaluate the integrals

$$I_n = \int_0^\pi \sin^2 nx \mathrm{d}x$$

e

$$I_{nm} = \int_0^\pi \sin nx \sin mx \mathrm{d}x$$

(c) Find the exact values of the coefficients a_n for $n \in \{1, 2, 3, 4, 5\}$ by evaluating

$$a_n = \frac{1}{I_n} \int_0^\pi x(\pi - x) \sin nx \mathrm{d}x$$

Why does this work? Compare your answers to the visual in- spection process.

1. Write a computer program for a Doppler sonar. The program should request the speed of sound, the frequency of the output pulse (or "ping"), the frequency of the reflected pulse, and the time delay between the output ping and the return ping. The program should then inform the user of the probable distance to the target and the target's possible speed(s) toward or away from the source. Try the program with the following data: the speed of sound is 340 m/s; the output pulse frequency is 20 kHz; the

frequency of the reflected pulse is 20.612 kHz; and the time delay between the output and reflected pings is 0.230 s. 2. Generalize the previous program so that the data from two consecutive pings can be used to determine both the distance to the target and the velocity of the target. The program will also need to request the rate at which outgoing pings are sent. Assume that the outgoing pings are omnidirectional, but the direction of incoming pings can be resolved. Try the program with the following data: the speed of sound is 340 m/s; the output pulse frequency is 20 kHz and the pulses are sent once per second; a 20.921 kHz reflected pulse coming from 40° E of N is received 0.288 s after the first pulse is sent; and a sec- ond 20.921 kHz reflected pulse coming from 36.5^o E of N is received 0.311 s after the second pulse is sent.

Capítulo 10

Fluidos

Problemas do Haliday Capítulo 14

1. (a) Show that the equations that govern the pressure as a function of the radial distance from the center of a spherical gaseous planet, in which the density is proportional to the pressure ($\rho = kp$), are $dp/dr = -(Gm/r^2)kp$ and $dm/dr = 4r^2kp$, where m is the mass contained within the sphere of radius r. (b) Numerically integrate these coupled equations outward from the point r_0 , where $r0 = 10^3m$, $p_0 = 2 \times 10^{16}Pa$, $m_0 = 7 \times 10^{14}kg$. Take the constant k to be $8 \times 10^12s^2/m^2$. Generate a graph of pressure against radial distance. (c) At what distance is the pressure less than one atmosphere?

A cylindrical water tank has a radius of 2 m and a height of 1.5 m. Originally the tank is completely filled with water, but a vertical crack appears in the tank and the water leaks out. Assuming the crack is 1 cm wide and extends from the base of the tank to the top, calculate the amount of time for the tank to completely empty. (Hint: Assume the crack is composed of 1 cm 2 holes, each one on top of the other, and solve the problem numerically.

Capítulo 11

Termodinâmica e Teoria Cinética dos Gases

Problemas do Haliday Capítulo 18 a 20

1. A soap bubble with surface tension $\gamma = 2.50 \times 10^2 N/m$ has a radius $r_0 = 2.0mm$ when the pressure outside the bub- ble is 1.0 atmosphere. (a) Numerically calculate the radius of the soap bubble when the pressure outside the bubble drops to 0.5 atm. (b) Numerically calculate the radius of the soap bub- ble if the pressure outside the bubble is raised to 2.0 atm. 2. A small balloon is filled with nitrogen gas (assumed ideal) at the bottom of the Marianas Trench, 35,000 ft beneath the sur- face of the ocean. The balloon originally has a radius of 1.0 mm, is massless, and is infinitely expandable without any sur- face tension, but always keeps a spherical shape. Assume the ideal gas inside the balloon is at 4^o C throughout this problem. The balloon begins to rise to the surface, as the balloon rises it expands, and as it moves there is a retarding force f propor- tional to speed v and balloon radius r given by

$$f = 6\pi\eta r v,$$

where $\eta = 1.7 \times 10^{-3} Ns/m$ is the viscosity of water. (a) Calculate the initial buoyant force on the balloon. (b) What will be the size of the balloon on the surface? (c) Numerically solve this problem to find out how long it takes for the bal- loon to rise to the surface.

1. Write a program to simulate the random walk of a particle. The particle starts at the origin, and can then take a step with Δx and Δy increments assigned randomly between -1 and $+1$. (a) Allow the particle to "walk" through 200 steps, and graph the motion. Choose the scale of the graph to just fit the data. (b) Allow the

particle to walk through 2000 steps, but this time plot the position of the particle only at the end of each 10 steps. Again, choose the scale of the graph to just fit the data. (c) Repeat, but now allow the particle to walk through 20,000 steps, and only plot the position at the end of each 100 steps. Compare the three graphs. How does the size of the graph grow with the number of steps? Do the graphs look similar? If the graphs were shuffled, would you be able to tell which was which?

2. Consider a van der Waals gas with $a = 0.10 Jm^3/mol$ and $b = 1.0 \times 10^4 m^3/mol$. (a) Find the temperature T_{cr} , pressure $p_c r$, and volume $V_c r$ where $p/V0$ and $2p/V20$. (b) Graph the pressure along isotherms as a function of vol- ume for 0.80T cr , 0.85T cr , 0.90T cr , 0.95T cr , 1.00T cr , 1.05T cr , and 1.10T cr . The graphs should extend from $V0$ to V 5V cr . (c) What is physically significant about the T cr isotherm?

Difusão da Molécula de um gás

Listing 11.1: Difusão da Molécula de Um Gás em 2D

```
import numpy as np
import matplotlib.pyplot as plt
import pylab as pl
from itertools import cycle


pi = np.pi
# Simular os passos
def random_walk(step_n, step_size):
    origin = np.zeros((1)) # comece em 0
    steps = 2*pi*np.random.uniform(low=0.0, high=1.0, size=step_n)
    x_steps = step_size*np.cos(steps)
    y_steps = step_size*np.sin(steps)
    x_path = np.concatenate([origin, x_steps]).cumsum(0)
    y_path = np.concatenate([origin, y_steps]).cumsum(0)
    return x_path, y_path


step_n = 100000
step_size = 1

x_path, y_path = random_walk(step_n, step_size)
xstart = x_path[:1]
ystart = y_path[:1]
xstop = x_path[-1:]
ystop = y_path[-1:]

fig = pl.figure(figsize=(8,4),dpi=200)
ax = fig.add_subplot(111)
ax.plot(x_path, y_path,c='blue',alpha=0.5,lw=0.5,ls='-');
ax.plot(xstart, ystart, c='red', marker='+')  #Plot o inicio
```

```
ax.plot(xstop, ystop, c='black', marker='o') # Plot o Final
pl.title('2D Random Walk')
pl.tight_layout(pad=0)
pl.grid(which="both")
pl.savefig('random_walk_1d.png',dpi=250);
```

Capítulo 12

O Campo Elétrico

Problemas do Haliday Capítulo 21 a 25

Calcule a força de atração entre dois anéis com cargas uniformemente distribuídas $+q$ e $-q$. O eixo dos anéis coincide com o eixo x e cada anel tem raio R.

Problema Repetir para um disco e para uma esfera

Um anel de raio $R = 1cm$ esta uniformemente carregado com carga Q. Um elétron se move no plano do anel. (a) Com $Q = -100\mu C$ Encontre a velocidade do elétron para que ele se mova em u

Capítulo 13

Circuitos Elétricos

Problemas do Haliday Capítulo 26 e 27

Capítulo 14

O Campo Magnético

Problemas do Haliday Capítulo 28, 29 e 30

Capítulo 15

Equações de Maxwell e Oscilações Eletromagnéticas

Problemas do Haliday Capítulo 30, 31, 32 e 33

Listing 15.1: Equação de Laplace

```
import numpy as np
import matplotlib.pyplot as plt

# Set maximum iteration
maxIter = 500

# Set rectangular Grid
lenX = lenY = 20
delta = 1

# Boundary condition
Vtop = 100
Vbottom = -100
Vleft = 0
Vright = 0

# Initial guess of interior grid
Vguess = 30

# Set colour interpolation and colour map.
# You can try set it to 10, or 100 to see the difference
# You can also try: colourMap = plt.cm.coolwarm
colorinterpolation = 50
colourMap = plt.cm.jet

# Set meshgrid
X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0, lenY))

# Set array size and set the interior value with Tguess
V = np.empty((lenX, lenY))
V.fill(Tguess)
```

```
# Set Boundary condition
V[(lenY-1):, :] = Vtop
V[:1, :] = Vbottom
V[:, (lenX-1):] = Vright
V[:, :1] = Vleft

# Iteration (We assume that the iteration is convergence in maxIter = 500)
print("Please wait for a moment")
for iteration in range(0, maxIter):
    for i in range(1, lenX-1, delta):
        for j in range(1, lenY-1, delta):
            V[i, j] = 0.25 * (V[i+1][j] + V[i-1][j] + V[i][j+1] + V[i][j-1])

print("Iteration finished")

# Configure the contour
plt.title("Contour of Eletric Potential")
plt.contourf(X, Y, V, colorinterpolation, cmap=colourMap)

# Set Colorbar
plt.colorbar()

# Show the result in the plot window
plt.show()
```

Capítulo 16

Óptica Física e Geométrica

Problemas do Haliday Capítulos 34 a 36

Capítulo 17

Princípio de Fermat e Outros Problemas de Extremização

Listing 17.1: Tempo de Viagem na braquistócrona e em outras curvas

```
import numpy as np
from scipy.optimize import newton
from scipy.integrate import quad
import matplotlib.pyplot as plt

# Acceleration due to gravity (m.s-2); final position of bead (m).
g = 9.81
x2, y2 = 1, 0.65

def cycloid(x2, y2, N=100):
    """Return the path of Brachistochrone curve from (0,0) to (x2, y2).

    The Brachistochrone curve is the path down which a bead will fall without
    friction between two points in the least time (an arc of a cycloid).
    It is returned as an array of N values of (x,y) between (0,0) and (x2,y2).

    """

    # First find theta2 from (x2, y2) numerically (by Newton-Rapheson).
    def f(theta):
        return y2/x2 - (1-np.cos(theta))/(theta-np.sin(theta))
    theta2 = newton(f, np.pi/2)

    # The radius of the circle generating the cycloid.
    R = y2 / (1 - np.cos(theta2))

    theta = np.linspace(0, theta2, N)
    x = R * (theta - np.sin(theta))
    y = R * (1 - np.cos(theta))

    # The time of travel
    T = theta2 * np.sqrt(R / g)
    print('T(cycloid) = {:.3f}'.format(T))
    return x, y, T
```

```
def linear(x2, y2, N=100):
    """Return the path of a straight line from (0,0) to (x2, y2)."""

    m = y2 / x2
    x = np.linspace(0, x2, N)
    y = m*x

    # The time of travel
    T = np.sqrt(2*(1+m**2)/g/m * x2)
    print('T(linear) = {:.3f}'.format(T))
    return x, y, T

def func(x, f, fp):
    """The integrand of the time integral to be minimized for a path f(x)."""

    return np.sqrt((1+fp(x)**2) / (2 * g * f(x)))

def circle(x2, y2, N=100):
    """Return the path of a circular arc between (0,0) to (x2, y2).

    The circle used is the one with a vertical tangent at (0,0).

    """

    # Circle radius
    r = (x2**2 + y2**2)/2/x2

    def f(x):
        return np.sqrt(2*r*x - x**2)
    def fp(x):
        return (r-x)/f(x)

    x = np.linspace(0, x2, N)
    y = f(x)

    # Calcualte the time of travel by numerical integration.
    T = quad(func, 0, x2, args=(f, fp))[0]
    print('T(circle) = {:.3f}'.format(T))
    return x, y, T

def parabola(x2, y2, N=100):
    """Return the path of a parabolic arc between (0,0) to (x2, y2).

    The parabola used is the one with a vertical tangent at (0,0).

    """

    c = (y2/x2)**2

    def f(x):
        return np.sqrt(c*x)
    def fp(x):
        return c/2/f(x)
```

```
    x = np.linspace(0, x2, N)
    y = f(x)

    # Calcualte the time of travel by numerical integration.
    T = quad(func, 0, x2, args=(f, fp))[0]
    print('T(parabola) = {:.3f}'.format(T))
    return x, y, T

# Plot a figure comparing the four paths.
fig, ax = plt.subplots()

for curve in ('cycloid', 'circle', 'parabola', 'linear'):
    x, y, T = globals()[curve](x2, y2)
    ax.plot(x, y, lw=4, alpha=0.5, label='{}: {:.3f} s'.format(curve, T))
ax.legend()

ax.set_xlabel('$x$')
ax.set_xlabel('$y$')
ax.set_xlim(0, 1)
ax.set_ylim(0.8, 0)
plt.savefig('brachistochrone.png')
plt.show()
```

Apêndice A

Elementos de *Python*

A.1 Mais informações sobre tipos de variáveis

A.2 Programação Orientada a Objetos em Python

Apêndice B

Python em aplicações numéricas: Numpy

B.1 Arrays

B.2 Métodos Estatísticos

B.3 Polinômios

B.4 Álgebra Linear

B.5 Sorteios Aleatórios

Calculando o valor de π. Ver a Seção E.5

Listing B.1: Estimando o valor de π usando sorteios aleatórios

```
import numpy as np

size = int(10**6)
experiments = int(10**3)

def pi_estimator(size):
    x = np.random.uniform(low = 0, high = 1, size = size)
    y = np.random.uniform(low = 0, high = 1, size = size)
    r = np.sqrt(x**2+y**2)
    pi_calculated = 0
    for R in r:
        if R<1:
            pi_calculated += 4/size
    return pi_calculated

pi_data = np.zeros(experiments)
for t in range(experiments):
    pi_data[t] = pi_estimator(size)
```

```
pi = np.mean(pi_data)
pi_error = np.var(pi_data)


print('\u03C0 =', pi, '\u00b1', pi_error)
print(pi-np.pi)
```

Passeio Aleatório

Listing B.2: Histogram da Distribuição de Probabilidade da caminhada Aleatória

```
import numpy as np
import matplotlib.pyplot as plt


def random_walk(step_set,prob, step_n):
    origin = np.zeros((1))
    step_shape = (step_n)
    steps = np.random.choice(a=step_set, size=step_shape, p = prob)
    path = np.concatenate([origin, steps]).cumsum(0)
    return path

step_n = 100000
step_set = [-1, 1]
p = 0.5
prob = [1-p, p]

simulations = 100000
data = np.zeros(simulations)
for t in range(simulations):
    data[t] = random_walk(step_set, prob, step_n)[-1:]
plt.hist(data, bins=100, density=True)
var = np.var(data)
mean = np.mean(data)
x = np.linspace(-1500,1500,500)
y = np.exp(-(x-mean)**2/(2*var))/np.sqrt(2*np.pi*var)
plt.plot(x,y, color='Red')
plt.show()
```

Apêndice C

Construção de Gráficos com Matplotlib

C.1 Mais obções básicas com gráficos

C.2 Histogramas

C.3 Plots de Contorno

C.4 Gráficos em 3D

C.5 Animações

Apêndice D

Computação Algébrica com Sympy

Apêndice E

Elementos de Cálculo Numérico

E.1 Soluções Numéricas

E.2 Integração Numérica

E.3 Diferenciação Numérica

E.4 Equações Diferenciais Ordinárias

E.4.1 Método de Euler

E.4.2 Variações do Método de Euler

E.4.3 Método de Runge-Kutta

E.5 Método de Monte Carlos

E.6 Métodos Estocásticos

E.7 Equações Diferenciais Parciais

E.7.1 Equação de Laplace

E.7.2 Equação da Onda

E.7.3 Equação do Calor

Apêndice F

Usando LaTeX

Referências Bibliográficas

[1] David Halliday, Robert Resnick, and Jearl Walker. Fundamentals of physics, enhanced problems version. *Fundamentals of Physics*, page 1128, 2002.

[2] Herch Moysés Nussenzveig. *Curso de física básica: Mecânica (vol. 1)*, volume 394. Editora Blucher, 2013.

[3] Herch Moysés Nussenzveig. *Curso de física básica: Ótica, relatividade, física quântica (vol. 4)*. Editora Blucher, 2014.

[4] Herch Moysés Nussenzveig. *Curso de física básica: Eletromagnetismo (vol. 3)*, volume 3. Editora Blucher, 2015.

[5] Herch Moysés Nussenzveig. *Curso de Física Básica: fluidos, oscilações e ondas, calor*, volume 2. Editora Blucher, 2018.

[6] Eric Ayars. Computational physics with python. *California State University*, 2013.

[7] Christian Hill. *Learning scientific programming with Python*. Cambridge University Press, 2020.

[8] Jeffrey C Lagarias. The 3 x+ 1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985.