Métodos Numéricos e Python no Ensino Médio

Edson Bertosa Lopes Santos
Luis Alberto D'Afonseca
Carlos Magno Martins Cosme

Métodos Numéricos e Python no Ensino Médio

Edson Bertosa Lopes Santos
Luis Alberto D'Afonseca
Carlos Magno Martins Cosme

Centro Federal dealign Educação Tecnológica de Minas Gerais – CEFET-MG
19 de dezembro de 2024

Esta apostila é produto do mestrado de Edson Bertosa Lopes Santos defendida em 2024 no Profmat do Cefet-MG.

A versão mais recente desta apostila pode ser baixada clicando ou escaneando o código QR.

Arte da capa: Fotografia de Designdrunkard baixada de Pexels

Sumário

Prefácio

Com imensa satisfação, apresentamos esta apostila dedicada à exploração dos métodos numéricos aplicados à resolução de desafios matemáticos com a utilização de recursos computacionais. Nosso objetivo é oferecer uma ferramenta versátil e eficaz para enriquecer sua jornada no mundo da matemática e da programação, independentemente de você ser um estudante entusiasta, um professor em busca de novas abordagens ou alguém curioso para explorar novos horizontes.

Como professor de matemática no Ensino Básico, percebo que os alunos frequentemente encontram dificuldades ao usar calculadoras para divisões e cálculos de raízes, como $1/3 \approx 0{,}333333$ ou $\sqrt{7} \approx 2{,}6457513$. Muitos ficam confusos com a precisão dos dígitos ou duvidam de seus próprios resultados. Isso me motivou a criar esta apostila, voltada para o estudo de métodos numéricos, que aborda técnicas para encontrar soluções aproximadas para problemas matemáticos onde cálculos diretos são inviáveis. Esses métodos permitem obter resultados suficientemente próximos dos valores corretos e são amplamente aplicáveis em várias áreas, como engenharia, física, economia e ciências da computação, oferecendo soluções práticas e eficientes para problemas complexos.

Nesta apostila, exploraremos desde os conceitos básicos de Python, uma linguagem de programação versátil e amplamente utilizada, até a aplicação de métodos numéricos para resolver problemas como zeros de funções e soluções de sistemas lineares e não lineares. Para melhor compreensão dos conceitos apresentados, é importante ter um conhecimento básico sobre funções, em nível de ensino médio. No entanto, não é necessário ter nenhum tipo de conhecimento prévio em programação, tornando o material acessível para alunos, professores e qualquer pessoa interessada em Matemática e Programação. A apostila oferece uma abordagem inclusiva e adaptável, com exemplos claros e exercícios práticos.

Agradecemos antecipadamente pelo seu interesse e dedicação em explorar conosco uma parte do vasto mundo da Matemática e da Programação . Venha descobrir

como a matemática e a tecnologia podem trabalhar em harmonia para solucionar problemas de forma eficaz e criativa!

1

Apresentação

Seja bem-vindo(a) à nossa apostila de introdução à linguagem de programação Python e aos métodos numéricos aplicados à resolução de problemas envolvendo zeros de funções e soluções de sistemas lineares e não lineares. Neste capítulo inicial, vamos apresentar de forma sucinta os temas abordados ao longo da apostila, material esse que pode contribuir para o desenvolvimento nessa área essencial, especialmente relevante nos dias atuais devido à crescente importância da tecnologia em diversos campos.

O Que é o Python e Por Que Estamos Trabalhando com Ele

O Python é uma linguagem de programação de alto nível, conhecida por sua simplicidade, legibilidade e versatilidade. É amplamente utilizada em diversas áreas, desde desenvolvimento de páginas para a internet até análise de dados e inteligência artificial.

Para começar a programar em Python, é necessário ter um ambiente de desenvolvimento configurado. Para isso vamos usar um aplicativo chamado QPython3L, para que seja possível programar no smartphone, ou uma plataforma online chamada Google Colab, que funciona tanto no smartphone como no computador, e não necessita de nenhum tipo de instalação.

No Capítulo 2, apresentaremos comandos básicos em Python, incluindo operações matemáticas, impressão de resultados e mensagens personalizadas, entrada e saída de dados, estruturas de repetição e controle de fluxo. Também desenvolveremos dois programas em Python, um que simula uma compra online, calculando o valor

total a ser pago por uma quantidade específica de produtos, e outro que gera a tabuada dinâmica de um número determinado inicialmente. Estes exemplos ajudarão a compreender como criar funções e utilizar estruturas de controle em Python. Posteriormente, exploraremos programas para resolver problemas matemáticos específicos, apresentando o código passo a passo e disponibilizando um link, que estará anexado ao código, para que o leitor possa acessá-lo diretamente na plataforma online do Google Colab.

Esa apostila não é um guia completo sobre Python. Para quem deseja se aprofundar mais na linguagem, recomendamos consultar o site oficial Python.org. Além disso, existem diversos cursos online disponíveis, como o Curso em Vídeo que pode ser uma excelente opção para expandir seus conhecimentos nessa linguagem de programação.

Funções

No Capítulo 3, vamos explorar o conceito de funções tanto na matemática quanto na programação. As funções desempenham um papel fundamental em ambos os campos, mas sua abordagem e aplicação podem variar significativamente.

Começaremos com uma revisão das funções na matemática, abordando conceitos como domínio, contradomínio, imagem, e definição formal de função. Veremos também diferentes tipos de funções, como lineares, quadráticas e trigonométricas, e como representá-las graficamente. Também falaremos sobre equações e um aspecto importante que é a busca por zeros de funções.

Em seguida, faremos uma transição para as funções na programação. Aqui, as funções são blocos de código que realizam tarefas específicas e podem ser reutilizadas em diferentes partes de um programa. Vamos aprender como definir funções em Python, passar argumentos para funções, e retornar valores de funções.

Finalmente, faremos uma comparação entre as funções na matemática e as funções na programação. Embora compartilhem o mesmo nome, as funções em cada domínio têm propósitos e características distintas. Compreender essas diferenças nos ajudará a usar as funções de forma eficaz em contextos matemáticos e de programação.

O Que São Métodos Numéricos Iterativos

Nos Capítulos 4 e 5, vamos explorar métodos numéricos iterativos. Um conjunto de técnicas que buscam encontrar a solução de um problema por meio de passos sucessivos, melhorando a aproximação a cada etapa. Esses métodos geralmente são utilizados quando não é possível encontrar uma solução direta através de métodos analíticos, permitindo-nos obter soluções aproximadas com a precisão desejada.

Nesta apostila, exploraremos uma variedade de métodos numéricos iterativos clássicos para solucionar problemas matemáticos. Usaremos o Método de Newton, o Método da Bisseção e o Método da Secante para a busca de raízes quadradas e zeros de funções, bem como o Método de Jacobi para sistemas lineares e novamente o Método de Newton para encontrar a solução de sistemas não lineares, explorando sua aplicação prática e sua importância no contexto matemático e computacional.

Estamos animados para começar essa jornada de aprendizado juntos. Vamos mergulhar no mundo da programação, métodos numéricos e descobrir como eles podem enriquecer o ensino da matemática!

2

Programação

Neste capítulo, embora não contenha uma introdução completa à programação, exploraremos de forma simplificada os comandos básicos da linguagem de programação Python, conhecida por sua simplicidade e versatilidade em diversas áreas, como ciência de dados, automação de tarefas, desenvolvimento web e de jogos, inteligência artificial, entre outras. Abordaremos as características que a torna uma ferramenta poderosa para resolver problemas matemáticos e científicos, mostrando de forma introdutória como escrever um programas e compreender estruturas de controle e dados para desenvolver algumas aplicações, podendo assim estimular a criatividade, o pensamento lógico e a resolução de problemas.

2.1 Plataformas de Programação

Nesta seção inicial, vamos apresentar as plataformas de programação que serão utilizadas nessa apostila. Após essa introdução às plataformas, seguiremos com os passos para o download, instalação e acesso às mesmas. Não se preocupe se você não tem experiência prévia em programação, este guia foi desenvolvido pensando em facilitar o aprendizado, fornecendo instruções claras e exemplos práticos, ajudando assim na compreensão dos processos executados.

Acesso pelo Smartphone

Para começar a programar em um smartphone, existem vários aplicativos disponíveis, tanto para IOS quanto para Android, tais como Python Code-Pad e Coding Python. No entanto, neste trabalho, optamos por utilizar o QPython3L. Uma plataforma Python gratuita e livre de anúncios, desenvolvida especialmente para dispositivos móveis Android. Uma vez instalado, o QPython3L pode ser utilizado sem a necessidade de conexão com a internet. Essa ferramenta é importante para facilitar o aprendizado da programação, especialmente considerando a grande intimidade que os jovens de hoje têm com seus smartphones. Ao ter acesso à programação através do dispositivo móvel, estamos conectando diretamente com o universo digital em que estamos imersos diariamente, podendo assim tornar o aprendizado mais acessível e dinâmico.

Vamos apresentar nesse momento os passos para fazer a instalação do aplicativo que será usado para escrever e executar o programa pelo smartphone.

1. Vá até a loja de apps do smartphone (Play Store - Android);
2. Procure pelo app 'QPython3L';
3. Clique em 'instalar' e após a instalação, clique em 'abrir';
4. Na página inicial do app, clique em 'Editor'.

A Figura 2.1 mostra todos os passos descritos anteriormente.

Figura 2.1: Instalação e acesso do aplicativo QPython3L.

Nesse momento, teremos uma página em branco onde vamos começar a escrever os códigos. Mas antes, é importante entender o que são código e *script* na programação, especialmente no contexto da linguagem Python.

O código refere-se às instruções escritas em uma linguagem de programação, como o Python, que são interpretadas ou compiladas para executar ações específicas no equipamento utilizado (computador, smatphones, entre outros). Essas instruções podem incluir operações matemáticas, manipulação de dados, controle de fluxo, entre outras funcionalidades. Já um *script*, por sua vez, é um conjunto de instruções ou comandos organizados em um arquivo de texto, geralmente com extensão ".py" no caso do Python. Esse arquivo contém o código que será executado pelo interpretador Python para realizar as tarefas programadas.

Antes de começarmos a escrever os códigos, vamos salvar nosso *script*, assim estamos guardando as instruções que escrevemos em um arquivo para que possamos executá-lo posteriormente e reutilizar o código desenvolvido. Essa prática é fundamental na programação, pois permite organizar e gerenciar projetos de forma eficiente. Para salvar o *script*, siga os passos:

1. Clique em 'salvar';
2. Clique no campo aberto para digitar o nome a ser dado para o arquivo que será salvo;
3. Escreva 'Teste.py';
4. Clique em 'ok'.

A Figura 2.2 mostra todos os passos descritos anteriormente.

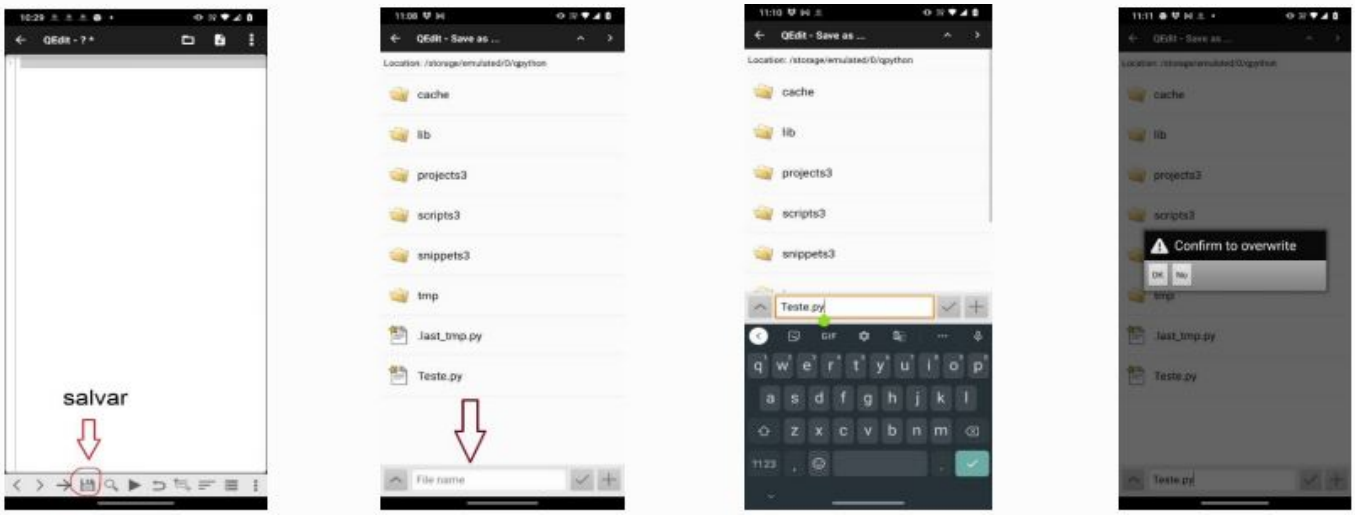

Figura 2.2: Salvando um *script* no aplicativo QPython3L.

Tendo executado os passos descritos, o *script* já estará salvo no smartphone. Você pode criar *scripts* para diferentes propósitos e nomeá-los como desejar. Para começar um novo *script*, com a pagina já aberta, siga os passos a seguir:

1. Click no símbolo +;
2. Click em 'script';
3. Digite 'Tabuada';
4. Click em 'ok'.

A Figura 2.3 mostra todos os passos descritos anteriormente.

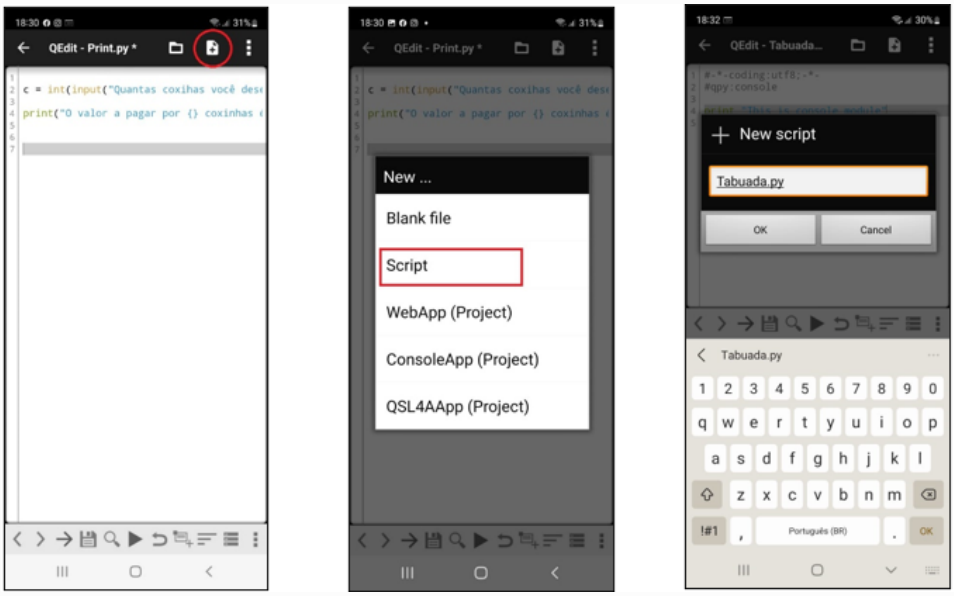

Figura 2.3: Criando um novo *script* no aplicativo QPython3L.

Nesse momento já estamos prontos para começar a programar através do app QPython3L no smartphone.

Acesso Online pelo Computador

Já para o uso online tanto para smartphones quanto para computadores, utilizaremos o Google Colab, uma plataforma que oferece um ambiente de desenvolvimento integrado para Python. A principal vantagem do Colab é a sua acessibilidade, pois não requer a instalação de nenhum software na máquina local, bastando apenas um aparelho com acesso à internet para começar a programar, o que o torna ideal para a sala de informática de uma escola, onde a disponibilidade de recursos técnicos pode variar. Essa abordagem online também promove a colaboração e o compartilhamento de projetos de forma fácil e eficiente.

Para acessar o Google Colab, basta ter uma conta do Google e acessar o site da plataforma pelo link Google Colab.

A Figura 2.4 mostra a tela inicial do Google Colab onde pode-se criar novos notebooks de Python ou abrir notebooks existentes diretamente no navegador.

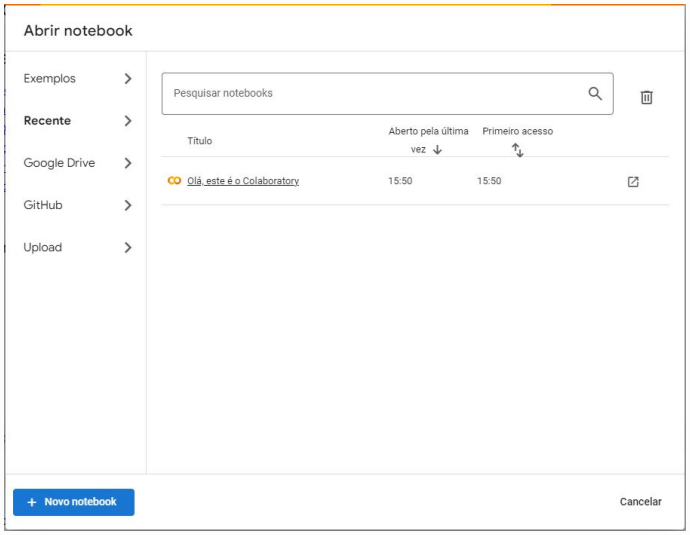

Figura 2.4: Página inicial do Google Colab.

Ao clicar no botão 'novo notebook' ou ao abrir um notebook já existente, teremos a plataforma onde serão escritos os códigos Python, mostrado na Figura 2.5 a seguir.

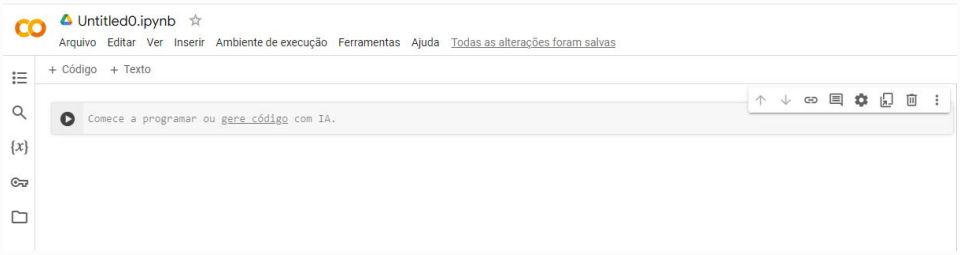

Figura 2.5: Plataforma de programação do Google Colab.

No Google Colab, você pode escrever código Python em células individuais. Para criar uma nova célula, clique no botão **'+ Codigo'** na barra de ferramentas. Em seguida, você pode digitar seu código Python na célula e executá-lo clicando no botão **'Run'** ao lado da célula.

Além de escrever e executar código Python, o Google Colab oferece diversos recursos adicionais, como suporte a bibliotecas populares, gráficos interativos, integração com

o Google Drive, entre outros. Você pode explorar esses recursos conforme avança em seus estudos de programação em Python.

Com esses conceitos básicos, você está pronto para começar a programar em Python utilizando o Google Colab.

Comandos Matemáticos Básicos

Os comandos matemáticos básicos em Python são essenciais para realizar operações numéricas. Abaixo, apresentamos uma tabela com alguns dos principais comandos.

Comando	**Descrição**
`+`	Adição
`-`	Subtração
`*`	Multiplicação
`/`	Divisão
`**`	Potenciação
`%`	Resto da divisão (módulo)
`=`	Atribuição de valor
`==`	Comparação de igualdade
`abs(x)`	Valor absoluto de x

Tabela 2.1: Comandos matemáticos básicos em Python

Esses comandos matemáticos básicos em Python formam a base para a realização de cálculos e manipulações numéricas em seus programas.

2.2 Entrada e Saída de Dados

Nesta seção, exploraremos os comandos básicos de entrada e saída de dados em Python de forma didática e gradual, detalhando todos os passos necessários para entender e aplicar esses conceitos. Iniciaremos com a construção de programas que interagem com o usuário, permitindo a coleta de informações e a exibição de resultados de maneira clara e organizada. Este processo facilita a compreensão desses conceitos fundamentais e pode capacitar o leitor a desenvolver habilidades práticas na criação de programas em Python, independentemente do seu nível de experiência em programação.

O Comando *print*

Vamos começar pelo comando `print`, sua função é mostrar um texto na tela. Vamos escrever um comando para que seja exposto na tela a frase "Olá mundo!". O comando deve incluir o texto entre aspas, que podem ser simples ou duplas. O texto entre aspas é chamado de *string* e o uso das aspas é necessário para que o texto apareça exatamente como foi digitado.

EXEMPLO 2.2.1: Escreva um código Python para exibir a frase "Olá mundo!" (código).

Vamos usar para isso o comando `print` que exibirá o que for escrito ou digitado dentro de aspas dentro de parênteses.

```
1 print("Olá mundo!")
```

Esse código gera o resultado apresentado a seguir.

```
Olá mundo!.
```

Vamos mostrar através do próximo exemplo a diferença entre um texto matemático escrito dentro de aspas e um escrito sem as aspas.

EXEMPLO 2.2.2: Escreva um código Python que mostre na tela a operação 5 + 5 na primeira linha e mostre o resultado da operação 5 + 5 na segunda linha (código).

Vamos usar para isso o comando `print`. Na primeira linha vamos colocar a operação dentro de aspas e na segunda linha vamos colocar a operação sem as aspas.

```
1 print("5+5")
2 print(5+5)
```

Esse código gera o resultado apresentado a seguir.

```
5+5
10
```

O programa exibiu `5 + 5` na primeira linha e `10` na segunda, que é o resultado da operação. Experimente outros exemplos, como `print(7+8)` ou `print(17-4+2)`, para

consolidar o aprendizado.

Agora, introduziremos conceitos algébricos utilizando variáveis como `x` e `y`, atribuindo a cada uma um valor numérico e realizando cálculos aritméticos com essas variáveis. Em Python, as variáveis funcionam como "caixinhas" onde armazenamos valores, que podem ser números, textos ou outros tipos de dados. É importante lembrar que as variáveis devem ser explicitamente atribuídas com um valor antes de serem usadas no programa.

EXEMPLO 2.2.3: Escreva um código Python para exibir o resultado da operação $x + y$, sendo $x = 2$ e $y = 3$ (código).

Vamos primeiramente atribuir os valores para as variáveis `x` e `y` e usar o comando `print` para exibir o resultado da operação.

```
1 x = 2
2 y = 3
3 print(x + y)
```

Esse código gera o seguinte resultado:

```
5
```

O programa exibiu na tela o número 5, resultado da operação $2 + 3$.

Nota: Se você não atribuir valores às variáveis `x` e `y` antes de tentar fazer a operação, o Python não saberá o que elas representam. Nesse caso, será gerado um erro chamado `NameError`, indicando que as variáveis não estão definidas. O erro será algo assim:

```
NameError: name 'x' is not defined
```

Portanto, é sempre necessário definir as variáveis com um valor antes de utilizá-las em operações.

Para o próximo exemplo vamos usar uma expressão um pouco mais complexa.

EXEMPLO 2.2.4: Escreva um código Python para exibir o resultado da operação $2a + 3b$ sendo $a = 4$ e $b = 7$ (código).

Vamos atribuir os valores para as variáveis `a` e `b` e usar o comando `print` para exibir o resultado da operação.

```
1 a = 4
2 b = 7
3 print( 2*a + 3*b)
```

Esse código gera o resultado apresentado a seguir.

```
29
```

O programa exibiu na tela o número 29, resultado de $2 \cdot 4 + 3 \cdot 7 = 8 + 21 = 29$. Se for do interesse do leitor, explore exemplos mudando os valores de `a` e `b` para que se consolide o aprendizado.

O próximo exemplo será mais prático. Vamos escrever um programa que mostra o valor a pagar de acordo com a quantidade de produtos que o cliente comprar.

EXEMPLO 2.2.5: Em uma lanchonete o preço de um salgado é de R$ 3,00. Escreva um programa que mostre o valor a pagar de acordo com a quantidade de salgados que o cliente comprar (código).

Vamos definir `n` como o número de salgados que o cliente vai comprar. Como cada salgado custa R$ 3,00, a expressão para o valor total a pagar será `3*n`. Supondo que o cliente compre 2 salgados, começaremos atribuindo o valor de 2 à `n` e usaremos o comando `print` para exibir o resultado da operação.

```
1 n = 2
2 print( 3*n )
```

Esse código gera o resultado apresentado a seguir.

```
6
```

O programa exibiu na tela o número 6, resultado de $3 \cdot 2 = 6$. Se for do interesse do leitor, explore mais exemplos mudando a quantidade de salgados comprados para que se consolide esse aprendizado.

O Comando *f-string*

Vamos explorar um comando importante em Python que facilita a formatação de saídas em programas: o comando `f-string`.

O comando `f-string` é utilizado para criar frases formatadas dinamicamente, permitindo inserir valores de variáveis diretamente em uma mensagem. Para usar esse comando, basta adicionar a letra `f` antes da *string* e incluir as variáveis dentro de chaves na mensagem. Vamos ver um exemplo para ilustrar como utilizar esse recurso.

EXEMPLO 2.2.6: Escreva um comando em Python que mostre na tela uma frase personalizada com o nome e a idade de uma pessoa (código).

Vamos criar as variáveis `nome` e `idade` e atribuir valores à elas, com o comando `f-string` colocaremos esses valores dentro do texto.

```
1 nome  = "João"
2 idade = 25
3
4 print(f"Olá, meu nome é {nome} e eu tenho {idade} anos.")
```

A saída desse código será:

```
Olá, meu nome é João e eu tenho 25 anos.
```

No exemplo acima, o `f-string` é utilizado para inserir dinamicamente os valores das variáveis `nome` e `idade` na frase, resultando em uma mensagem personalizada. Se trocarmos o valor dado às variáveis, a estrutura da frase não será alterada.

O Comando *input*

Vamos explorar um outro importante comando em Python que facilita a interação com o usuário: o comando `input`.

O comando `input` permite interagir diretamente com o usuário, solicitando que ele insira textos ou valores. Para lidar com números inteiros, use `int` para converter a entrada; para números com casas decimais, use `float`. Vamos ver um exemplo de como utilizar esse comando.

Exemplo 2.2.7: Escreva um código Python onde o programa irá solicitar o nome, a idade e a altura (em metros) do usuário e, ao ser executado, mostrará na tela uma frase personalizada com as informações recebidas (código).

Vamos iniciar o código solicitando ao usuário que digite seu nome, idade e altura, usando o comando `input` para obter esses valores e armazená-los nas variáveis `nome`, `idade` e `altura`. Em seguida, o programa exibirá uma mensagem de boas-vindas contendo essas informações. Para isso, utilizaremos o comando `print` em conjunto com o comando `f-string` para formatar a mensagem.

```
1 print("Digite seu nome e sua idade por favor.")
2 nome = input("Digite seu nome: ")
3 idade = int(input("Digite sua idade: "))
4 altura = float(input("Digite sua altura: "))
5 print(f"Olá, meu nome é {nome}, tenho {idade} anos e tenho
     {altura} metros de altura.")
```

Assumindo que o usuário digite o nome Alberto, para a idade o número inteiro 29 e para a altura o número 1,72, esse código gera o resultado apresentado a seguir.

```
Digite seu nome e sua idade por favor.
nome: Alberto
idade: 29
altura: 1.72
Olá, meu nome é Alberto, tenho 29 anos e
tenho 1.72 metros de altura.
```

Cada comando neste programa desempenha uma função específica. O comando `print` é utilizado para exibir mensagens, solicitando informações ao usuário. O comando `input` captura a entrada de dados, como nome, idade e altura, e retorna uma string. Usamos `int` e `float` para informar o programa sobre o tipo de atribuição que será dada para a variável, que no caso é um número inteiro e um número decimal respectivamente. Por fim, `print` exibe a mensagem final, incorporando as variáveis com a formatação `f-string`.

Com isso, finalizamos esta seção. Na próxima seção, apresentaremos alguns comandos básicos de controle de fluxo, que são essenciais para o desenvolvimento de programas em Python.

2.3 Repetições e Controle de Fluxo

Nesta seção, vamos explorar os comandos básicos de controle de fluxo em Python, como estruturas de decisão (`if`, `elif`, `else`) e estruturas de repetição (`for`, `while`). Esses comandos são fundamentais para criar programas mais dinâmicos e flexíveis, permitindo que o código tome decisões e execute ações repetidamente com base em condições específicas. Vamos aprender como usar essas estruturas de maneira eficiente para resolver problemas comuns de programação e criar programas mais complexos e interativos.

Os Comandos *list* e *range*

Antes de começar com os comandos citados, é importante conhecer um comando que auxilia na criação e execução de programas que envolvem repetições: o comando `range`. Esse comando gera sequências de números, o que é especialmente útil em *loops* e iterações. O `range` pode ser utilizado com um, dois ou três argumentos para criar intervalos variados:

- `range(fim)`: Gera uma sequência de 0 até `fim - 1`.
- `range(inicio, fim)`: Gera uma sequência de `inicio` até `fim - 1`.
- `range(inicio, fim, passo)`: Gera uma sequência de `inicio` até `fim - 1`, com intervalos de `passo`.

Vamos explorar alguns exemplos para entender o funcionamento desse comando (código).

```
1 print(range(6))
2 print(range(3, 10))
3 print(range(1, 20, 3))
```

A resposta do programa será:

```
range(0, 6)
range(3, 10)
range(1, 20, 3)
```

O comando `range` não gera uma lista com os elementos. Para utilizar a sequencia de números gerados, devemos armazená-los em uma lista e, em seguida, usar os comandos de manipulação, como o `print` para imprimir os elementos.

O comando `list` em Python é um tipo de variável que converte uma sequência de itens em uma lista, que é uma coleção ordenada e mutável de elementos de tipos variados, como números, *strings* e outras listas. A sintaxe básica é `list(sequencia)`, onde `sequencia` representa os itens a serem convertidos.

Vamos usar o comando `list` junto com o comando `range` para criar e exibir uma lista com os números de uma sequência (código).

```
1 print(list(range(6)))
2 print(list(range(3, 10)))
3 print(list(range(1, 20, 3)))
```

A resposta do programa será:

```
1 [0, 1, 2, 3, 4, 5]
2 [3, 4, 5, 6, 7, 8, 9]
3 [1, 4, 7, 10, 13, 16, 19]
```

Os programas geralmente não são escritos com essa estrutura; é mais comum atribuir valores a variáveis e trabalhar com elas. Veja o mesmo programa reescrito utilizando variáveis (link).

```
1 r1 = range(6)
2 r2 = range(3, 10)
3 r3 = range(1, 20, 3)
4
5 s1 = list(r1)
6 s2 = list(r2)
7 s3 = list(r3)
8
9 print(s1)
10 print(s2)
11 print(s3)
```

A resposta do programa será:

```
1 [0, 1, 2, 3, 4, 5]
2 [3, 4, 5, 6, 7, 8, 9]
3 [1, 4, 7, 10, 13, 16, 19]
```

- A sequência `s1` gera uma sequência de 0 até 5, porque o `range(6)` inclui números de 0 até 5 (6 é excluído).
- A sequência `s2` gera números de 3 até 9, porque o `range(3, 10)` começa em 3 e vai até 9 (10 é excluído).
- A sequência `s3` gera números de 1 até 19 (20 é excluído) com intervalos de 3, porque o `range(1, 20, 3)` começa em 1 e a cada passo adiciona 3 até alcançar ou ultrapassar 19. Qualquer valor acima de 19 será excluído.

Esses exemplos mostram de forma simplificada o funcionário dos comandos `list` e `range`, porém as listas são amplamente utilizadas na linguagem de programação Python e possuem diversas formas de serem utilizadas ou modificadas. Para mais informações sobre esse comando, uma opção pode ser o vídeo Usando Listas do canal Curso em Vídeo ou buscando informações no site oficial Python.org.

O Comando *for*

O comando `for` em Python é usado para criar um *loop* que itera sobre uma sequência de números. Ele é útil quando é necessário executar um bloco de código um número específico de vezes. Por exemplo, para criar a tabuada do 2, poderíamos usar um comando já estudado na Seção 2.2, o comando `print` e optar por escrever o código a seguir.

```
1 print(2*1)
2 print(2*2)
3 print(2*3)
4 print(2*4)
5 print(2*5)
6 print(2*6)
7 print(2*7)
8 print(2*8)
9 print(2*9)
10 print(2*10)
```

Teríamos na tela todos os resultados da tabuada do 2. Porém, dá muito trabalho repetir todos esses comandos para todos os números da tabuada. Para reduzir tarefas que exigem a repetição de ações similares, usamos o comando `for`. A sintaxe básica é

```
for variavel in range(inicio, fim, passo):,
```

onde `variavel` é a variável que assume o valor de cada item na sequência a cada iteração do *loop*, `inicio` é o número inicial da sequência (incluído no *loop*), `fim` é o

número final da sequência (excluído do loop) e `passo` é o intervalo entre os números na sequência (padrão é 1 se não especificado).

Vamos aplicar o comando `for` para substituir todos os comandos `print` digitados anteriormente para conseguirmos os valores da tabuada do número 2.

EXEMPLO 2.3.1: Escreva um código Python que mostra todos os resultados da tabuada do número 2 (código).

O comando `for x in range(1,11):` faz com que o comando digitado logo abaixo seja repetido e a variável `x` receba os valores de 1 até 10, pois o último valor (11) não é atribuído. Para obtermos os resultados da tabuada do número 2, o código será

```
1 for x in range(1, 11):
2     print(2*x)
```

Esse código gera o resultado apresentado a seguir.

```
2
4
6
8
10
12
14
16
18
20
```

Observe que com esse comando conseguimos todos os resultados da tabuada do 2 de forma mais simples. Esse é um dos comandos principais da linguagem Python.

Para que a tabuada seja mostrada de forma mais completa, o ideal seria ser exposta na forma $2 \times 1 = 2$. Para isso vamos usar o comando `f-string` para mostrar a tabuada de forma mais completa. Veja o exemplo a seguir.

EXEMPLO 2.3.2: Escreva um código Python que mostra a tabuada do número 2 de forma completa (código).

Vamos usar o comando `for x in range` para atribuir os valores à variavel e em seguida vamos usar o comando `print` e o comando `f-string` para que o programa exiba a tabuada formatada.

```
1 for x in range(1,11):
2     print(f"2 x {x} = {2*x}")
```

Esse código gera o resultado apresentado a seguir.

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

O resultado será exatamente como descrevemos. Para ver a tabuada de outros números basta substituir o número 2 pelo número desejado.

Esses exemplos apresentaram de forma simplificada o comando `for`, porém podemos utilizá-lo de diversas formas diferentes. Para mais informações sobre esse comando, uma opção pode ser o vídeo Estrutura de repetição for do canal Curso em Vídeo ou buscar informações no site oficial Python.org.

O Comando *if*

O comando `if` funciona como um filtro que permite a execução de determinadas ações apenas se uma condição específica for verdadeira. É como uma bifurcação na estrada: se a condição for atendida, o programa segue por um caminho específico, executando o bloco de código dentro do `if`; caso contrário, ele continua por outro caminho. Por exemplo, podemos usar o `if` para verificar se um número é positivo ou negativo e executar diferentes ações com base nessa verificação.

Exemplo 2.3.3: Crie um programa em Python que verifica se um número digitado pelo usuário é positivo, negativo ou zero (código).

Vamos utilizar o comando `input` para que o usuário digite um número, o comando `if` para verificar a condição e o comando `print` para imprimir uma mensagem correspondente.

```
numero = int(input("Digite um número: "))

if numero > 0:
    print("O número é positivo.")
elif numero < 0:
    print("O número é negativo.")
else:
    print("O número é zero.")
```

Supondo que o usuário digite o número -3, a resposta do programa será

```
Digite um número: -3
O número é negativo.
```

Nesse código, `input` solicita um número inteiro ao usuário. O comando `if` verifica se o número é maior que zero e imprime "O número é positivo." Se não, `elif` verifica se é menor que zero e imprime "O número é negativo." Caso contrário, `else` imprime "O número é zero." O comando `print` exibe a mensagem correspondente.

Este exemplo mostra de forma simplificada o funcionamento do comando `if`. Para mais informações sobre esse comando, uma opção pode ser o vídeo Condições (Parte 1) do canal Curso em Vídeo ou busque informações no site oficial Python.org.

O Comando *while*

O comando `while` em Python é como uma porta que permanece aberta enquanto uma condição específica for verdadeira. Isso significa que o código dentro do bloco de `while` será repetido continuamente até que a condição não seja mais atendida, momento em que o programa segue para o próximo bloco de código após o `while`. Vamos ver um exemplo de aplicação desse comando.

EXEMPLO 2.3.4: Crie um programa em Python que pede ao usuário para adivinhar um número de 1 a 5. O programa continua solicitando uma entrada até que o usuário acerte o número correto, finalizando com uma mensagem (código).

Vamos usar o *loop* `while` para continuar pedindo ao usuário para adivinhar um número até que ele digite o número correto (neste caso, 3). Uma mensagem de sucesso é exibida quando o usuário acerta.

```
numero_secreto = 3
adivinhacao    = 0

while adivinhacao != numero_secreto:
    adivinhacao = int(input("Adivinhe um número de 1 a 5: "))
    if adivinhacao != numero_secreto:
        print("Tente novamente!")

print("Parabéns! Você adivinhou o número correto.")
```

Supondo que o usuário digitou como tentativas os números 2, 4 e 3, a resposta do programa será

```
Adivinhe um número de 1 a 5: 2
Tente novamente!
Adivinhe um número de 1 a 5: 4
Tente novamente!
Adivinhe um número de 1 a 5: 3
Parabéns! Você adivinhou o número correto.
```

Este código utiliza um *loop* `while` para continuar solicitando ao usuário que adivinhe um número de 1 a 5 até que o número correto seja digitado. O bloco de `if` verifica se a variável `adivinhacao` é diferente do `numero_secreto` e, se for, imprime uma mensagem para tentar novamente. Quando a variável `adivinhacao` se iguala ao `numero_secreto`, o *loop* termina e uma mensagem de parabéns é exibida.

Este exemplo mostra de forma simplificada o funcionamento do comando `while`. Para mais informações sobre esse comando, uma opção pode ser o vídeo Estrutura de repetição while do canal Curso em Vídeo ou busque informações no site oficial Python.org.

2.4 Consolidando o Aprendizado com Exemplos Práticos

Para consolidar o que aprendemos até aqui, vamos criar dois programas práticos que integrarão todos os comandos e conceitos que exploramos até agora. Estes exemplos finais têm o objetivo de mostrar como os comandos básicos de entrada e saída de dados, operadores matemáticos, comandos de repetição e de controle de fluxo podem ser aplicados em situações do dia a dia e resolver problemas concretos.

Tabuada Eletrônica Interativa

Se retomarmos o programa que gera a tabuada de um número usando o comando `for`, conforme descrito em 2.3, podemos melhorar a interação com o usuário ao incluir o comando `input`. Isso permitirá que o usuário digite o número da tabuada desejada. Após exibir a tabuada, podemos utilizar os comandos `if` e `while` para perguntar se o usuário gostaria de ver a tabuada de outro número. Veja o exemplo a seguir.

EXEMPLO 2.4.1: Crie um programa em Python que solicite um número ao usuário e mostre a tabuada desse número. Após a execução do programa, pergunte ao usuário se deseja ver a tabuada de outro número. Se a resposta for sim, o programa será executado novamente; caso contrário, será exibida uma mensagem de encerramento (código).

Vamos chamar de `n` o número do qual o usuário deseja ver a tabuada. Usaremos um `while` para criar um *loop*. O comando `input` solicitará ao usuário o número desejado. O comando `for` criará a tabuada e, após mostrá-la na tela, o programa perguntará se o usuário deseja ver outra tabuada, novamente usando `input`. Se a resposta for "Sim", o programa reinicia. Caso contrário, o programa imprime "Programa encerrado." e sai do *loop*, encerrando a execução.

É importante notar a diferença entre o símbolo de atribuição `=` e o símbolo de comparação `==`: o primeiro é utilizado para atribuir um valor a uma variável, enquanto o segundo é usado para comparar se dois valores são iguais.

```
1 resposta = "Sim"
2
3 while resposta == "Sim":
```

```
4     n = int(input("De qual número você deseja ver a sua
    tabuada? "))
5
6     for x in range(1, 11):
7         print(f"{n} x {x} = {n*x}")
8
9     resposta = input("Deseja ver a tabuada de outro número? ")
10
11 print("Programa encerrado.")
```

O programa inicia perguntando ao usuário de qual número deseja-se ver a tabuada. Assumindo que o usuário deseja ver a tabuada do número 7 e em seguida a tabuada do número 9, o resultado do código apresentado será.

```
De qual número você deseja ver a sua tabuada? 7

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70

Deseja ver a tabuada de outro número? Sim
De qual número você deseja ver a sua tabuada? 9

9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
```

```
Deseja ver a tabuada de outro número? Não
Programa encerrado.
```

Agora podemos ver a tabuada do número inicialmente digitado e após, podemos digitar um outro número qualquer que o programa vai gerar uma nova tabuada.

Programa de Compra Online

Nosso exemplo consiste em criar um programa de compra online. O programa informa ao usuário o valor do produto e solicita o número desejado. Após calcular o total a pagar, o programa solicita a confirmação da compra.

EXEMPLO 2.4.2: Crie um programa em Python que informe ao usuário o valor do produto e solicite o número desejado. O programa deve calcular e exibir o valor total a ser pago e pedir a confirmação da compra (código).

Vamos utilizar o comando `input` para capturar o número de produtos desejados, calcular o total multiplicando o número de produtos pelo preço unitário com `*`, e exibir o resultado usando `print`. Em seguida, vamos utilizar `input` novamente para solicitar a confirmação da compra. Vamos utilizar `if-else` para analisar a resposta do usuário.

```
1  continuar = "Sim"
2  while continuar == "Sim":
3      print("O produto custa 12 reais cada.")
4      n= int(input("Quantos produtos o Sr(a) deseja comprar? "))
5
6      total = 12 * n
7      print(f"Total a pagar por {n} produtos: {total} reais.")
8
9      resposta = input("Deseja confirmar a compra? (Sim/Não): ")
10
11     if resposta == "Sim":
12         print("Compra confirmada. Obrigado!")
13         continuar = "Não"
14     else:
15         print("Compra não confirmada. Reiniciando o
       processo.")
```

Assumindo que o usuário insira inicialmente o número 10 e depois mude para 6 antes de confirmar a compra, o código gera o resultado apresentado abaixo:

```
O produto custa 12 reais cada.
Quantos produtos o Sr(a) deseja comprar? 10
Total a pagar por 10 produtos: 120 reais.
```

```
Deseja confirmar a compra? (Sim/Não): Não
Compra não confirmada. Reiniciando o processo.

O produto custa 12 reais cada.
Quantos produtos o Sr(a) deseja comprar? 6
Total a pagar por 6 produtos: 72 reais.
Deseja confirmar a compra? (Sim/Não): Sim
Compra confirmada. Obrigado!
```

Temos agora um programa que simula uma compra online, informando ao usuário o valor do produto e solicitando o número desejado. Após, calcula o total a pagare solicita a confirmação da compra.

Ao final deste capítulo sobre programação em Python, concluímos nossa jornada desde os comandos de "Entrada e Saída de Dados", onde exploramos conceitos fundamentais como declaração de variáveis e operações aritméticas, até comandos para "Repetições e Controle de Fluxo", que nos permitiu praticar estruturas de repetição e a lógica algorítmica. Nesses programas usamos alguns comandos básicos e de extrema importância para a programação na linguagem Python, outros comandos Python serão apresentados nos capítulos seguintes e também vamos discutir sobre funções e como criar um programa para solucionar alguns problemas matemáticos específicos.

3

Funções

Neste capítulo, exploraremos funções, tanto na Matemática quanto na programação em Python. As funções são uma parte essencial do entendimento do mundo ao nosso redor, permitindo-nos descrever padrões, modelar fenômenos naturais e resolver problemas complexos.

Começaremos com uma revisão de funções na Matemática, abordando conceitos como domínio, contradomínio, imagem e definição formal de função. Em seguida, exploraremos funções afins, quadráticas, exponenciais e trigonométricas, discutindo algumas de suas propriedades e apresentando seus gráficos no plano cartesiano. Após essa revisão, abordaremos equações e zeros de funções, explorando algumas técnicas utilizadas para encontrar os valores de x que satisfazem $f(x) = 0$. Em seguida, faremos a transição para funções em Python, que são blocos de código reutilizáveis para executar tarefas específicas em diferentes partes de um programa.

Ao comparar as funções na Matemática com as funções em Python, entenderemos melhor as semelhanças e diferenças entre esses dois domínios, ampliando nossa capacidade de usar funções de forma eficiente em contextos matemáticos e de programação.

3.1 Funções na Matemática

Uma função é uma relação matemática que associa elementos de um conjunto de origem (domínio) à elementos de um conjunto de destino (contradomínio). Ela descreve como cada elemento do domínio está relacionado a um único elemento do contradomínio. Por exemplo, vamos considerar uma função f que associa mercadorias à seus preços.

$$f(\text{Mercadoria}) = \text{Preço}.$$

Aqui está uma tabela mostrando alguns exemplos de mercadorias e seus preços associados.

Mercadoria(kg)	**Preço (R$)**
Maçã	7,49
Banana	4,79
Laranja	2,49
Arroz	4,99
Açúcar	6,29
Feijão	7,49

Nesta tabela, cada mercadoria do domínio está associada a um único preço no contradomínio, ou seja, uma mercadoria não pode ter dois valores diferentes. Isso significa que cada item está em um e somente um preço, enquanto no contradomínio os preços podem se repetir, como por exemplo, a maçã e o feijão têm preços iguais a R$ 7,49. Além disso, a imagem da função f, que é o conjunto de todos os preços que a função pode produzir a partir dos itens do domínio, pode ser diferente do contradomínio, nesse exemplo o contradomínio eram todos os preços possíveis, porém a imagem são somente os preços mostrados na tabela.

Formalmente, uma função $f\colon A \to B$ é uma regra que associa a cada elemento x do conjunto A um único elemento y do conjunto B. O domínio A é o conjunto de todos os valores possíveis para x, enquanto o contradomínio B é o conjunto de todos os valores possíveis para y. A imagem da função f, denotada por $\text{Im}(f)$ ou $f(A)$, é o conjunto de todos os valores que f produz a partir dos elementos de A.

Esses conceitos são essenciais para compreendermos como as funções operam e como podemos utilizar suas propriedades para resolver problemas matemáticos de forma eficiente e precisa. Vale ressaltar que as funções podem representar fenômenos complexos do mundo real, como a valorização de ações na bolsa ao longo do tempo ou a relação da quantidade de chuva em uma cidade com as estações do ano. Esses exemplos evidenciam a versatilidade das funções e sua aplicabilidade na modelagem de diversos fenômenos reais.

Função Afim

Uma função afim é da forma

$$f(x) = ax + b,$$

onde a e b são constantes reais, sendo $a \neq 0$. Como x pode receber qualquer valor real, a função representa uma reta no plano cartesiano, onde a determina a inclinação da reta em relação ao eixo horizontal (eixo x) e b indica o ponto de interseção com o eixo vertical (eixo y).

A Figura 3.1 apresenta o gráfico da função $f(x) = 2x + 3$ no plano cartesiano (código).

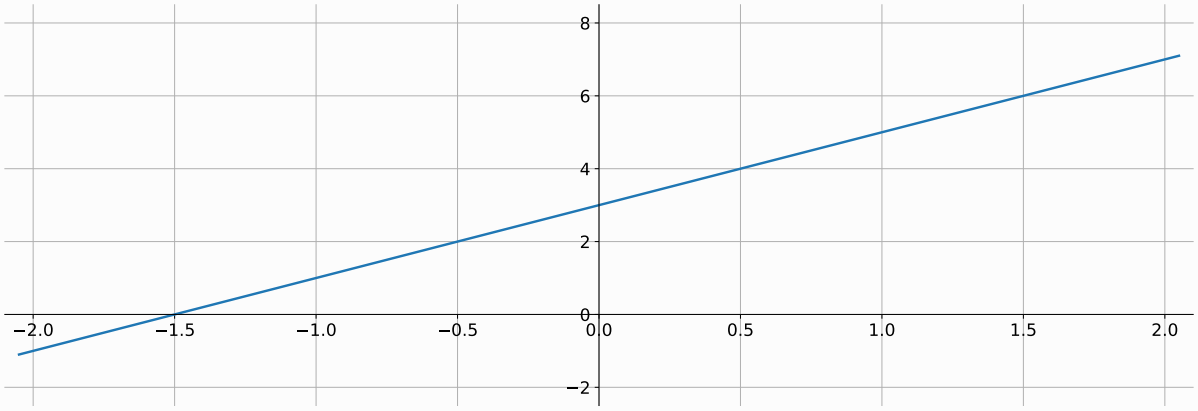

Figura 3.1: Gráfico da função $f(x) = 2x + 3$ no plano cartesiano.

Esse gráfico mostra que se trata de uma reta com inclinação positiva de 2 unidades para cada unidade no eixo x, e intersecta o eixo y em 3. Para determinar os pontos pertencentes ao gráfico dessa função, basta atribuir valores específicos para x e realizar os cálculos correspondentes. Vamos fazer isso no exemplo a seguir.

Exemplo 3.1.1: Determine alguns pontos pertencentes ao gráfico da função $f(x) = 2x + 3$.

Para determinar alguns pontos pertencentes ao gráfico da função $f(x) = 2x + 3$, vamos calcular os valores de $f(x)$ para $x = -2, -1, 0, 1, 2$.

$$\begin{aligned} f(-2) &= 2(-2) + 3 = -1 \\ f(-1) &= 2(-1) + 3 = 1 \\ f(0) &= 2 \cdot 0 + 3 = 3 \\ f(1) &= 2 \cdot 1 + 3 = 5 \\ f(2) &= 2 \cdot 2 + 3 = 7 \end{aligned}$$

Portanto, os pontos $(-2, -1)$, $(-1, 1)$, $(0, 3)$, $(1, 5)$ e $(2, 7)$ pertencem ao gráfico da função $f(x) = 2x + 3$.

Podemos criar um programa em Python para calcular os valores de $f(x)$ como mostra o exemplo a seguir.

Exemplo 3.1.1: Escreva um programa em Python para calcular os valores de $f(x) = 2x + 3$ para para alguns valores específicos de x (código).

Vamos escrever um código para calcular os valores da função $f(x) = 2x + 3$ para $x = -2, -1, 0, 1$ e 2. O código será:

```
1 # Atribuindo valores para x.
2 x_valores = [-2, -1, 0, 1, 2]
3
4 # Efetuando os cálculos correspondentes.
5 for x in x_valores:
6     f_x = 2*x + 3
7     print(f"f({x}) = {f_x}")
```

Este código calcula os valores de $f(x)$ para cada valor de x especificado na lista `x_valores` e imprime os resultados. A resposta do programa será:

```
1 f(-2) = -1
2 f(-1) =  1
3 f(0)  =  3
4 f(1)  =  5
5 f(2)  =  7
```

Assim, podemos ver como a função $f(x) = 2x + 3$ é avaliada para diferentes valores de x usando Python.

Função Quadrática

A função quadrática

$$f(x) = ax^2 + bx + c,$$

onde a, b e c são constantes reais e $a \neq 0$. O gráfico dessa função é uma parábola no plano cartesiano e está é utilizada para modelar diversos fenômenos naturais e artificiais. Vamos explorar as propriedades das funções quadráticas baseado e seus coeficientes.

A concavidade da parábola é determinada pelo coeficiente a. Se $a > 0$, a parábola tem concavidade para cima, e se $a < 0$, a parábola tem concavidade para baixo e c indica o ponto de interseção com o eixo vertical (eixo y).

As Figuras 3.2 e 3.3 mostram o gráfico de duas funções quadráticas, uma com concavidade para cima ($a > 0$) e outra com concavidade para baixo ($a < 0$).

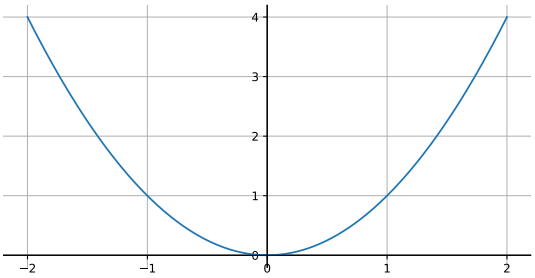

Figura 3.2: Concavidade para cima $(a > 0)$.

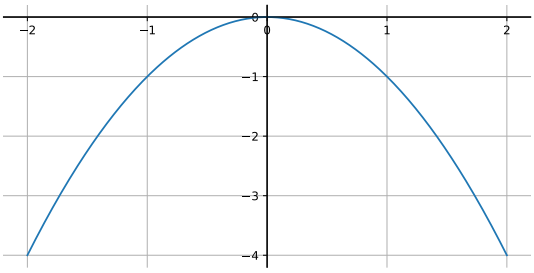

Figura 3.3: Concavidade para baixo $(a < 0)$.

A Figura 3.4 mostra o gráfico da função $f(x) = x^2 - 6x + 5$ no plano cartesiano (código).

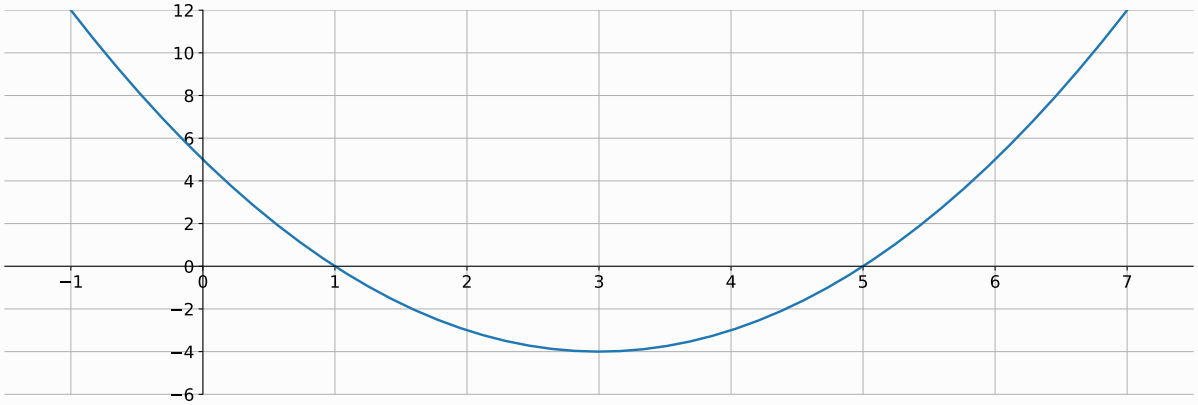

Figura 3.4: Gráfico da função $f(x) = x^2 - 6x + 5$ no plano cartesiano.

Esse gráfico mostra que se trata de uma parábola com concavidade voltada para cima, e intersecta o eixo y em 5. Vamos determinar alguns pontos pertencentes a esse gráfico atribuindo valores específicos para x e realizando os cálculos correspondentes. Vamos fazer isso no exemplo a seguir.

EXEMPLO 3.1.2: Determine alguns pontos pertencentes ao gráfico da função $f(x) = x^2 - 6x + 5$.

Para determinar alguns pontos pertencentes ao gráfico da função $f(x) = x^2 - 6x + 5$, vamos calcular os valores de $f(x)$ para $x = -2, -1, 0, 1, 2$.

$$\begin{aligned} f(-2) &= (-2)^2 - 6(-2) + 5 = 21 \\ f(-1) &= (-1)^2 - 6(-1) + 5 = 12 \\ f(0) &= 0^2 - 6 \cdot 0 + 5 = 5 \\ f(1) &= 1^2 - 6 \cdot 1 + 5 = 0 \\ f(2) &= 2^2 - 6 \cdot 2 + 5 = -3 \end{aligned}$$

Portanto, os pontos $(-2, 21)$, $(-1, 12)$, $(0, 5)$, $(1, 0)$ e $(2, -3)$ pertencem ao gráfico da função $f(x) = x^2 - 6x + 5$.

Podemos criar um programa em Python para calcular os valores de $f(x)$ como mostra o exemplo a seguir.

Exemplo 3.1.2: Escreva um programa em Python para calcular os valores de $f(x) = x^2 - 6x + 5$ para para alguns valores específicos de x (código).

Vamos escrever um código para calcular os valores da função $f(x)$ para $x = -2, -1, 0, 1, 2, 3, 4, 5$. O código será:

```
# Atribuindo valores para x.
x_valores = [-2, -1, 0, 1, 2, 3, 4, 5]

# Efetuando os cálculos correspondentes.
for x in x_valores:
    f_x = x**2 - 6*x + 5
    print(f"f({x}) = {f_x}")
```

Os resultados serão:

```
f(-2) =  21
f(-1) =  12
 f(0) =  5
 f(1) =  0
 f(2) = -3
 f(3) = -4
 f(4) = -3
 f(5) =  0
```

Assim, podemos ver como a função $f(x) = x^2 - 6x + 5$ é avaliada para diferentes valores de x usando Python.

Função Exponencial

As funções exponenciais do tipo $f(x) = ab^x$, onde $b > 0$ e $b \neq 1$, são importantes na matemática e na modelagem de fenômenos como crescimento populacional, decaimento radioativo e propagação de epidemias. O coeficiente a controla a amplitude da função, determinando $f(0)$ e influenciando a altura da curva em relação ao eixo x. A base b define o comportamento da função: quando $b > 1$, $f(x)$ apresenta crescimento exponencial conforme x aumenta; quando $0 < b < 1$, a função representa um decaimento exponencial. As Figuras 3.5 e 3.6 mostra os gráficos das funções $f(x) = 2^x$ e $f(x) = \left(\frac{1}{2}\right)^x$, ilustrando esses comportamentos (código).

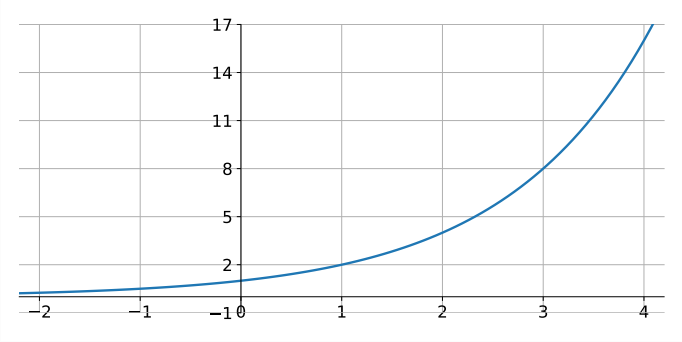

Figura 3.5: Gráfico da função $f(x) = 2^x$.

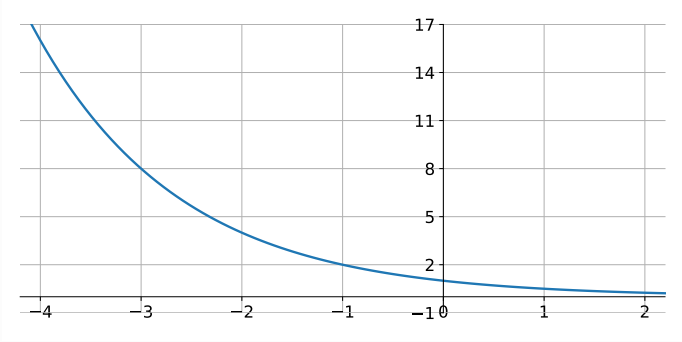

Figura 3.6: $f(x) = (1/2)^x$.

Essas funções exponenciais são amplamente utilizadas na modelagem matemática de fenômenos naturais, financeiros e científicos, sendo uma ferramenta importante em áreas como economia, biologia, física, engenharia, entre outras.

Vamos ver um exemplo considerando a função $f(x) = 2 \cdot 3^x$. A Figura 3.7 apresenta o gráfico dessa função no plano cartesiano (código).

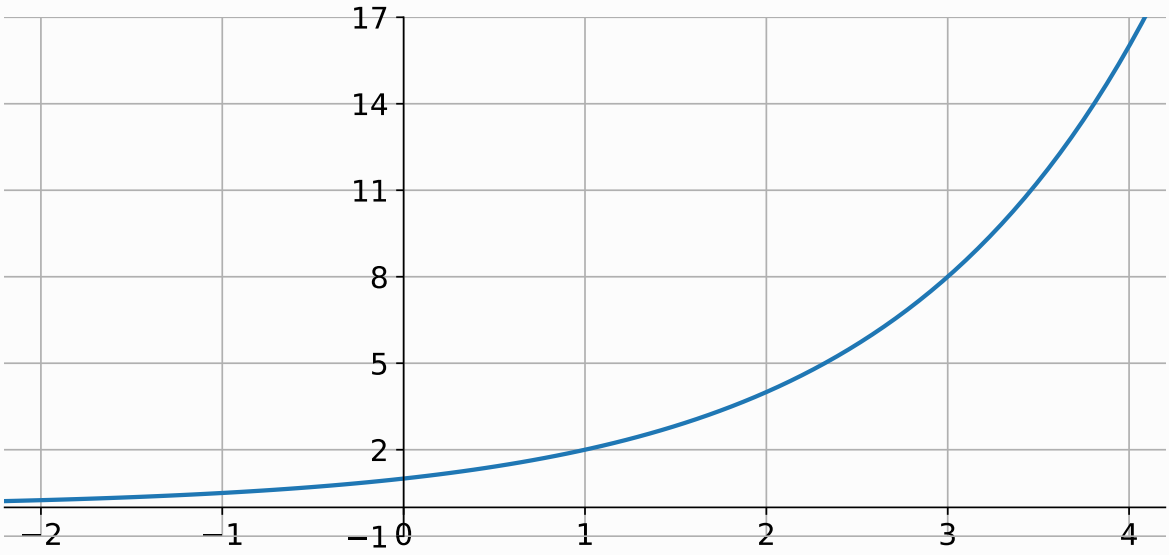

Figura 3.7: Gráfico da função $f(x) = 2 \cdot 3^x$ no plano cartesiano.

O gráfico representa um crescimento exponencial à medida que os valores de x aumentam. Vamos determinar alguns pontos pertencentes a esse gráfico atribuindo valores específicos para x e realizando os cálculos correspondentes. Vamos fazer isso no exemplo a seguir.

EXEMPLO 3.1.3: Determine alguns pontos pertencentes ao gráfico da função $f(x) = 2 \cdot 3^x$.

Para determinar alguns pontos pertencentes ao gráfico da função $f(x) = 2 \cdot 3^x$, vamos calcular os valores de $f(x)$ para $x = -1,\ 0,\ 1, 2,\ 3$.

$$\begin{aligned} f(-1) &= 2 \cdot 3^{-1} = \frac{2}{3} \\ f(0) &= 2 \cdot 3^0 = 2 \\ f(1) &= 2 \cdot 3^1 = 6 \\ f(2) &= 2 \cdot 3^2 = 18 \\ f(3) &= 2 \cdot 3^3 = 54 \end{aligned}$$

Portanto, os pontos $\left(-1, \frac{2}{3}\right)$, $(0, 2)$, $(1, 6)$, $(2, 18)$ e $(3, 54)$ pertencem ao gráfico da função $f(x) = 2 \cdot 3^x$.

Podemos criar um programa em Python para calcular os valores de $f(x)$ como mostra o exemplo a seguir.

EXEMPLO 3.1.3: Escreva um programa em Python para calcular os valores de $f(x) = 2 \cdot 3^x$ para para alguns valores específicos de x (código).

Vamos escrever um código para calcular os valores da função $f(x)$ para $x = -1, 0, 0.25, 1, 1.49, 2, 3$. O código será:

```
1 # Atribuindo valores para x.
2 x_valores = [-1, 0, 0.25, 1, 1.49, 2, 3]
3
4 # Efetuando os cálculos correspondentes.
5 for x in x_valores:
6     f_x = 2 * 3**x
7     print(f"f({x}) = {f_x}")
```

Os resultados serão:

```
1 f(-1)   = 0.6666
2 f(0)    = 2
3 f(0.25) = 2.6321
4 f(1)    = 6
5 f(1.49) = 10.278
```

```
6 f(2)    = 18
7 f(3)    = 54
```

Aqui, podemos observar claramente uma das vantagens da programação. Determinar os valores da função $f(x)$ para $x = 0{,}25$ ou $x = 1{,}49$ não seria uma tarefa simples de realizar manualmente. Utilizando técnicas de programação, conseguimos obter esses valores de forma rápida e precisa, destacando a eficiência da computação em resolver problemas matemáticos complexos.

Funções Trigonométricas

As funções trigonométricas, como o seno (sen), o cosseno (cos) e a tangente (tg), desempenham um papel importante na matemática, ligadas aos ângulos e ao círculo trigonométrico. Elas ajudam a compreender o comportamento periódico dessas funções e suas relações nos diferentes quadrantes. Além de serem importantes na geometria dos triângulos, essas funções são essenciais para a análise de oscilações e fenômenos periódicos em diversas áreas da matemática e das ciências.

Função Seno

A função seno (sen) é uma função trigonométrica que relaciona um ângulo à um comprimento de um lado em um triângulo retângulo. Especificamente, o seno de um ângulo agudo é definido como a razão entre o comprimento do cateto oposto a esse ângulo e o comprimento da hipotenusa. Essa definição é expandida ao círculo unitário, onde o seno de um ângulo é dado pela coordenada y do ponto correspondente no círculo, que possui raio igual a 1.

A função seno é periódica, com um período de 2π , o que implica que $\text{sen}(\theta + 2\pi) = \text{sen}(\theta)$ para qualquer valor de θ . Além disso, essa função varia entre -1 e 1 , sendo amplamente utilizada na modelagem de fenômenos cíclicos, como ondas sonoras e eletromagnéticas, bem como em aplicações em física, engenharia e outras disciplinas.

Sua forma gráfica é uma onda suave que cruza o eixo horizontal em múltiplos de π .

A Figura 3.8 apresenta o gráfico da função $f(x) = \text{sen}(x)$ no plano cartesiano (código).

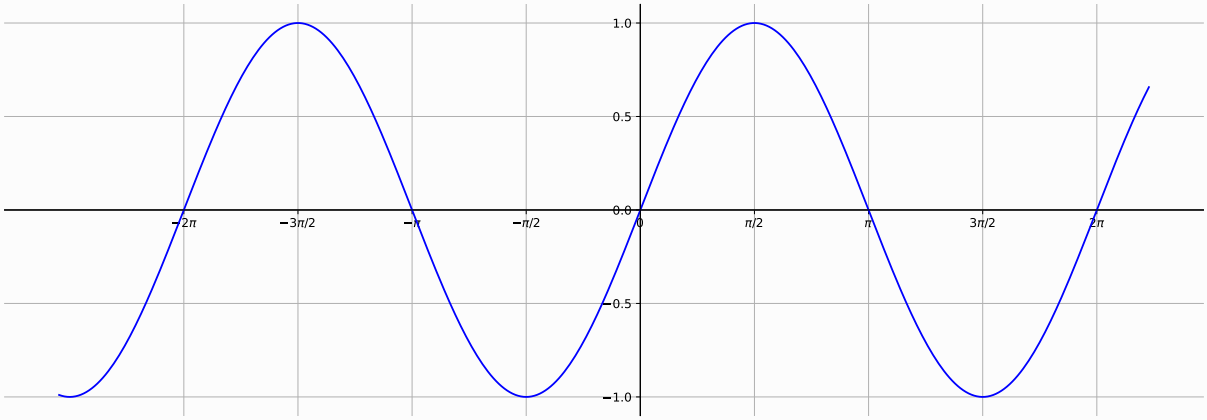

Figura 3.8: Gráfico da função $f(x) = \text{sen}(x)$ no plano cartesiano.

No gráfico da função $f(x) = \text{sen}(x)$, observa-se um padrão periódico de oscilação conforme o valor de x varia. Vamos determinar alguns pontos pertencentes a esse gráfico atribuindo valores específicos para x e realizando os cálculos correspondentes.

EXEMPLO 3.1.4: Calcule o valor da função $f(x) = \text{sen}(x)$ para alguns valores determinados de x.

Consideremos a função $f(x) = \text{sen}(x)$. Podemos calcular os valores da função para alguns ângulos. Vamos fazer isso para os seguintes ângulos no intervalo de 0 a $\frac{\pi}{2}$:

$$\begin{aligned} f(0) &= \text{sen}(0) &&= 0 \\ f\left(\frac{\pi}{6}\right) &= \text{sen}\left(\frac{\pi}{6}\right) &&= \frac{1}{2} \\ f\left(\frac{\pi}{4}\right) &= \text{sen}\left(\frac{\pi}{4}\right) &&= \frac{\sqrt{2}}{2} \\ f\left(\frac{\pi}{3}\right) &= \text{sen}\left(\frac{\pi}{3}\right) &&= \frac{\sqrt{3}}{2} \\ f\left(\frac{\pi}{2}\right) &= \text{sen}\left(\frac{\pi}{2}\right) &&= 1 \end{aligned}$$

Portanto, os pontos $(0, 0)$, $\left(\frac{\pi}{6}, \frac{1}{2}\right)$, $\left(\frac{\pi}{4}, \frac{\sqrt{2}}{2}\right)$, $\left(\frac{\pi}{3}, \frac{\sqrt{3}}{2}\right)$, $\left(\frac{\pi}{2}, 1\right)$ pertencem ao gráfico da função $f(x) = \text{sen}(x)$.

Agora veremos um código em Python para calcular os valores da função $f(x) = \text{sen}(x)$

para alguns valores determinados de x. Para isso vamos utilizar uma biblioteca específica para cálculos matemáticos mais avançados, a biblioteca `math`.

Em Python, nem todas as funcionalidades estão disponíveis por padrão. Algumas, como funções matemáticas avançadas, precisam ser importadas de bibliotecas externas. Bibliotecas em Python são conjuntos de ferramentas que oferecem funcionalidades específicas, permitindo aos programadores escreverem códigos de maneira mais eficiente. Para utilizá-las, é necessário importá-las com a palavra-chave `import` seguida do nome da biblioteca.

Nesta seção, vamos importar a biblioteca `math`, que fornece acesso a várias funções matemáticas, como trigonometria, exponenciais e logaritmos, facilitando cálculos mais complexos.

Função	**Descrição**
`math.sqrt(x)`	Retorna a raiz quadrada de x
`math.pow(x, y)`	Retorna x elevado a y
`math.sin(x)`	Retorna o seno de x (em radianos)
`math.cos(x)`	Retorna o cosseno de x (em radianos)
`math.tan(x)`	Retorna a tangente de x (em radianos)
`math.log(x)`	Retorna o logaritmo natural de x

Tabela 3.1: Comandos básicos da biblioteca `math`

Vamos ver o exemplo a seguir.

EXEMPLO 3.1.4: Escreva um programa em Python para calcular os valores de $f(x) = \text{sen}(x)$ para para alguns valores específicos de x (código).

Vamos escrever um código para calcular os valores da função $f(x)$ para $x = -1$, 0, 0.3, 1, 1.78, 2, 3. O código será:

```
1 # Importando a biblioteca math
2 import math
3
4 # Atribuindo valores para x
5 angulos = [-1, 0, 0.3, 1, 1.78, 2, 3]
6
7 # Efetuando os cálculos correspondentes
8 for angulo in angulos:
9     seno_angulo = math.sin(angulo)
```

```
10     print(f"sen({angulo}) = {seno_angulo}")
```

Os resultados serão:

```
1 sen(-1)   = -0.8414
2 sen(0)    = 0.0
3 sen(0.3)  = 0.2955
4 sen(1)    = 0.8414
5 sen(1.78) = 0.9781
6 sen(2)    = 0.9092
7 sen(3)    = 0.1411
```

Dessa forma, podemos observar como a função $f(x) = \text{sen}(x)$ é avaliada para diferentes valores de x, utilizando Python.

Função Cosseno

A função cosseno (cos) é uma função trigonométrica que relaciona um ângulo a uma razão geométrica. No contexto do círculo trigonométrico, o cosseno de um ângulo é definido como a coordenada x do ponto correspondente no círculo unitário. Assim, para um ângulo θ, temos $\cos(\theta) = x$, onde x é a projeção do ponto no círculo sobre o eixo x.

A função cosseno é periódica, com período 2π, o que significa que $\cos(\theta+2\pi) = \cos(\theta)$ para qualquer valor de θ. A imagem desta função está no intervalo entre -1 e 1, apresentando um comportamento similar ao da função seno, e é amplamente utilizada em diversas áreas, como física e engenharia, para modelar fenômenos que envolvem ciclos e oscilações.

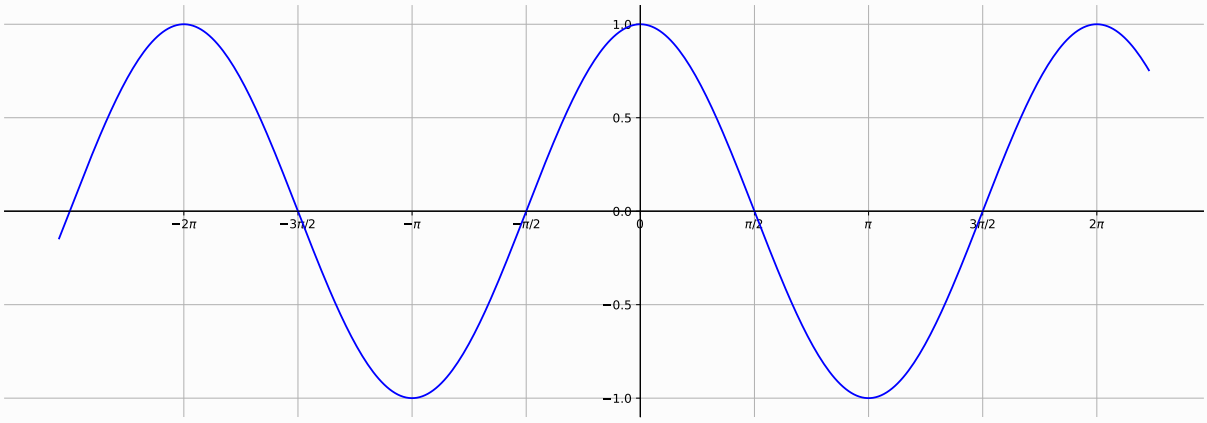

Figura 3.9: Gráfico da função $f(x) = \cos(x)$ no plano cartesiano.

A Figura 3.9 apresenta o gráfico da função $f(x) = \cos(x)$ no plano cartesiano (código).

No gráfico da função $f(x) = \cos(x)$, observa-se um padrão periódico de oscilação conforme o valor de x varia. Vamos determinar alguns pontos pertencentes a esse gráfico atribuindo valores específicos para x e realizando os cálculos correspondentes.

EXEMPLO 3.1.5: Calcule o valor da função $f(x) = \cos(x)$ para alguns valores determinados de x.

Consideremos a função $f(x) = \cos(x)$. Podemos calcular os valores da função para alguns ângulos. Vamos fazer isso para os seguintes ângulos no intervalo de 0 a 2π:

$$
\begin{aligned}
f(0) &= \cos(0) &&= 1 \\
f\left(\frac{\pi}{6}\right) &= \cos\left(\frac{\pi}{6}\right) &&= \frac{\sqrt{3}}{2} \\
f\left(\frac{\pi}{4}\right) &= \cos\left(\frac{\pi}{4}\right) &&= \frac{\sqrt{2}}{2} \\
f\left(\frac{\pi}{3}\right) &= \cos\left(\frac{\pi}{3}\right) &&= \frac{1}{2} \\
f\left(\frac{\pi}{2}\right) &= \cos\left(\frac{\pi}{2}\right) &&= 0
\end{aligned}
$$

Portanto, os pontos $(0, 1)$, $\left(\frac{\pi}{6}, \frac{\sqrt{3}}{2}\right)$, $\left(\frac{\pi}{4}, \frac{\sqrt{2}}{2}\right)$, $\left(\frac{\pi}{3}, \frac{1}{2}\right)$, $\left(\frac{\pi}{2}, 0\right)$ pertencem ao gráfico da função $f(x) = \cos(x)$.

Agora, vamos ver um código em Python para calcular os valores da função $f(x) = \cos(x)$ utilizando a biblioteca `math`.

EXEMPLO 3.1.5: Escreva um programa em Python para calcular os valores de $f(x) = \cos(x)$ para para alguns valores específicos de x (código).

Vamos escrever um código para calcular os valores da função $f(x) = \cos(x)$ para $x = -1$, 0, 0.3, 1, 1.78, 2, 3. O código será:

```
1 # Importando a biblioteca math
2 import math
```

```
3
4 # Atribuindo valores para x
5 angulos = [-1, 0, 0.3, 1, 1.78, 2, 3]
6
7 # Efetuando os cálculos correspondentes
8 for angulo in angulos:
9     cosseno_angulo = math.cos(angulo)
10    print(f"cos({angulo}) = {cosseno_angulo}")
```

Os resultados serão:

```
1 cos(-1)   = 0.5403
2 cos(0)    = 1.0
3 cos(0.3)  = 0.9553
4 cos(1)    = 0.5403
5 cos(1.78) = -0.2076
6 cos(2)    = -0.4161
7 cos(3)    = -0.9899
```

Dessa forma, podemos observar como a função $f(x) = \cos(x)$ é avaliada para diferentes valores de x, utilizando Python.

Função Tangente

A função tangente (tg) é uma das funções trigonométricas fundamentais. Ela é definida como a razão entre o seno e o cosseno de um ângulo. Matematicamente, podemos expressar a tangente da forma

$$\text{tg}(x) = \frac{\text{sen}(x)}{\cos(x)},$$

onde $x \neq \frac{\pi}{2} + k\pi$, com $k \in \mathbb{Z}$. Isso ocorre porque a tangente não está definida para esses valores de x, onde o cosseno é igual a zero, resultando em uma divisão por zero. A função tangente é periódica, com período π, e varia de $-\infty$ a $+\infty$.

Essa definição é válida tanto no contexto de um triângulo retângulo quanto no círculo trigonométrico. No triângulo retângulo, a tangente representa a razão entre o comprimento do cateto oposto e o comprimento do cateto adjacente ao ângulo. No círculo trigonométrico, ela é a razão entre as coordenadas y (seno) e x (cosseno) do ponto correspondente ao ângulo sobre a circunferência.

No plano cartesiano, o gráfico da função tangente apresenta características distintas, como suas assíntotas verticais e a periodicidade da função. A Figura 3.10 apresenta o gráfico da função $f(x) = \text{tg}(x)$ no intervalo de 0 a 2π (código) .

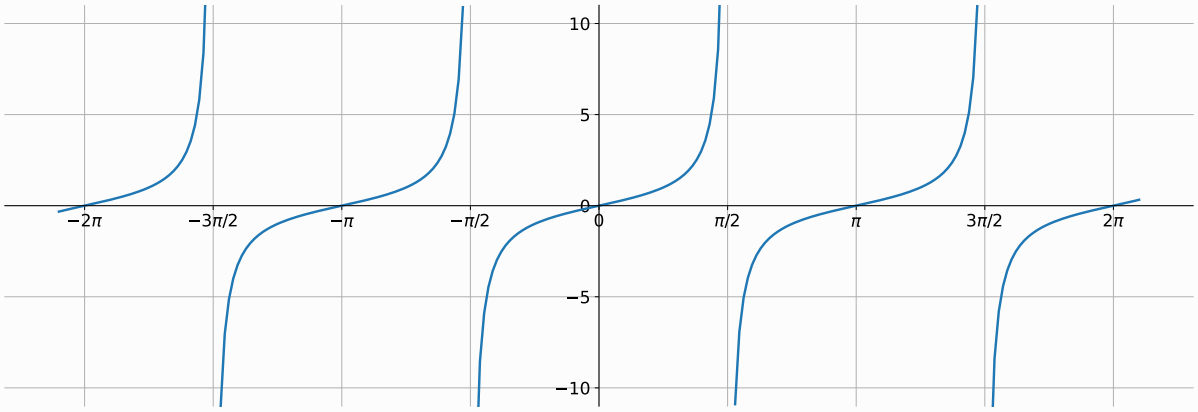

Figura 3.10: Gráfico da função $f(x) = \text{tg}(x)$ no plano cartesiano.

No gráfico da função $f(x) = \text{tg}(x)$, observe as assíntotas verticais nos pontos onde $x = \dfrac{\pi}{2} + k\pi$, indicando os pontos onde a tangente é indefinida. Vamos determinar alguns pontos pertencentes a esse gráfico atribuindo valores específicos para x e realizando os cálculos correspondentes.

EXEMPLO 3.1.6: Calcule o valor da função $f(x) = \text{tg}(x)$ para alguns valores determinados de x.

Consideremos a função tangente $f(x) = \text{tg}(x)$. Podemos calcular os valores da função para alguns ângulos. Vamos fazer isso para os seguintes ângulos no intervalo de 0 a $\dfrac{\pi}{2}$:

$$\begin{aligned}
f(0) = \text{tg}(x) &= 0 \\
f\left(\frac{\pi}{6}\right) = \text{tg}\left(\frac{\pi}{6}\right) &= \frac{\sqrt{3}}{3} \\
f\left(\frac{\pi}{4}\right) = \text{tg}\left(\frac{\pi}{4}\right) &= 1 \\
f\left(\frac{\pi}{3}\right) = \text{tg}\left(\frac{\pi}{3}\right) &= \sqrt{3} \\
f\left(\frac{\pi}{2}\right) = \text{tg}\left(\frac{\pi}{2}\right) &= \text{indefinido}
\end{aligned}$$

Portanto, os pontos $(0,0)$, $\left(\frac{\pi}{6}, \frac{\sqrt{3}}{3}\right)$, $\left(\frac{\pi}{4}, 1\right)$, $\left(\frac{\pi}{3}, \sqrt{3}\right)$ pertencem ao gráfico da função $\text{tg}(x)$.

Agora, vamos ver um código em Python para calcular os valores da função $f(x) = \text{tg}(x)$ utilizando a biblioteca `math`.

EXEMPLO 3.1.6: Escreva um programa em Python para calcular os valores de $f(x) = \text{tg}(x)$ para alguns valores específicos de x (código).

Vamos escrever um código para calcular os valores da função $f(x)$ para $x = -1$, 0, 0.3, 1, 1.78, 2, 3. O código será:

```
# Importando a biblioteca math
import math

# Atribuindo valores para x
angulos = [-1, 0, 0.3, 1, 1.78, 2, 3]

# Efetuando os cálculos correspondentes
for angulo in angulos:
    tangente_angulo = math.tan(angulo)
    print(f"tg({angulo}) = {tangente_angulo}")
```

Os resultados serão:

```
tg(-1)   = -1.557
tg(0)    = 0.0
tg(0.3)  = 0.3093
tg(1)    = 1.5574
tg(1.78) = -4.7100
tg(2)    = -2.1850
tg(3)    = -0.1425
```

Dessa forma, podemos observar como a função $f(x) = \text{tg}(x)$ é avaliada para diferentes valores de x utilizando Python.

Para concluir, as funções trigonométricas, como o seno, cosseno e tangente, são fundamentais na matemática e nas ciências devido à sua capacidade de modelar oscilações e padrões periódicos que aparecem em diversos fenômenos naturais e

artificiais. Essas funções não apenas ajudam a compreender a geometria dos triângulos e a circularidade dos ângulos, mas também são essenciais para a análise de ondas, sinais e movimentos cíclicos. Utilizando ferramentas computacionais como Python e suas bibliotecas, como `math`, podemos calcular, visualizar e explorar essas funções de maneira eficiente, permitindo em muitos casos uma análise detalhada e precisa de comportamentos angulares e periódicos em várias áreas do conhecimento, desde a física até a engenharia e além.

3.2 Equações e Zeros de Funções

Uma equação é uma sentença matemática aberta, uma afirmação matemática que expressa a igualdade entre duas expressões. De forma geral, uma equação envolve uma ou mais variáveis, e encontrar as soluções da equação significa determinar os valores dessas variáveis que tornam a igualdade verdadeira. Quando há apenas uma variável envolvida, a equação pode ser escrita na forma $f(x) = 0$, onde $f(x)$ é uma função matemática que depende da variável x. Os valores de x que satisfazem essa equação são chamados de zeros da função $f(x)$, pois são os pontos onde o gráfico de $f(x)$ intercepta o eixo x. Em outras palavras, resolver uma equação é equivalente a encontrar os zeros da função associada a essa equação. A seguir, exploraremos alguns métodos para encontrar esses zeros e resolver equações de maneira eficiente.

Existem diversos métodos para encontrar zeros de funções, cada um com suas características e aplicabilidades. Vamos abordar dois desses métodos.

Método Analítico: Neste método, escrevemos $f(x) = 0$ e resolvemos a equação resultante para encontrar a solução ou as soluções. Existem equações em que pode ser bastante complicado ou até mesmo pode não ser possível encontrar uma solução de forma analítica.

Método Numérico: Os métodos numéricos abrangem diversas técnicas para aproximar soluções de problemas matemáticos complexos ou difíceis de serem solucionados analiticamente. Dentro desta categoria, os métodos iterativos utilizam algoritmos que repetem cálculos até encontrar uma solução satisfatória, começando com uma estimativa inicial e refinando-a a cada iteração. Esses métodos são especialmente úteis quando soluções analíticas são inviáveis ou complicadas, e dependem de computadores para realizar cálculos de forma eficiente e precisa. Portanto, enquanto os métodos numéricos são amplos, os métodos iterativos se focam em refinar estimativas até alcançar a solução desejada.

Nesse capítulo, vamos explorar os métodos analíticos para encontrar zeros de algumas funções. No Capítulo 4, vamos abordar métodos numéricos para encontrar os zeros de uma função específica, fornecendo mais detalhes e explicações.

Para ilustrar os métodos analíticos, consideremos exemplos com diferentes tipos de funções.

EXEMPLO 3.2.1: Encontre o zero da função afim $f(x) = 2x - 4$ de forma analítica.

Escrevendo a equação na forma $f(x) = 0$, para encontrar essa solução podemos executar os passos a seguir.

$$\begin{aligned} f(x) &= 0 \\ 2x - 4 &= 0 \\ 2x &= 4 \\ x &= \frac{4}{2} \\ x &= 2 \end{aligned}$$

Assim encontrando a solução da equação que é $x = 2$, que se trata de um zero da função $f(x) = 2x - 4$, pois $f(2) = 2 \cdot 2 - 4 = 0$. Uma Função Afim admite apenas um valor como sendo um zero da função.

A Figura 3.11 apresenta o gráfico da função $f(x) = 2x - 4$ destacando seu zero (código).

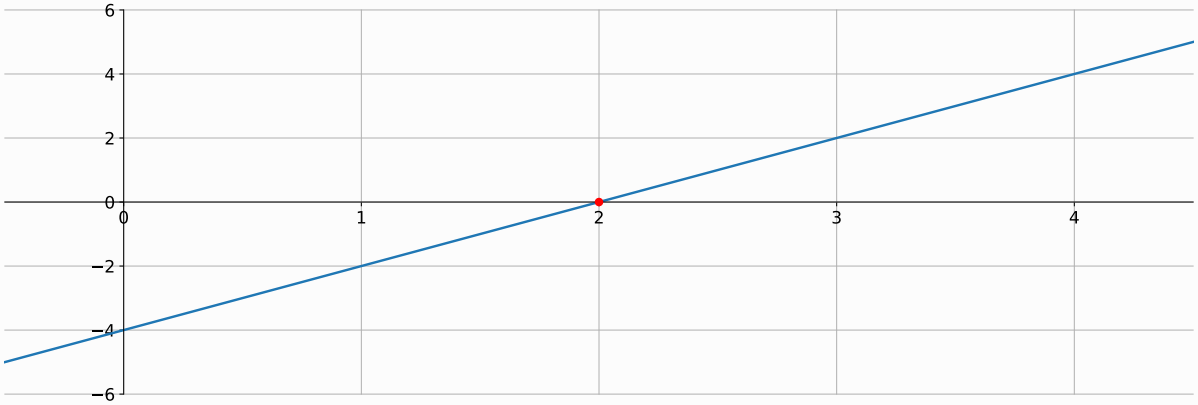

Figura 3.11: Gráfico de $f(x) = 2x - 4$ com seu zero destacado.

Vamos agora analisar as funções quadráticas. No caso do método analítico, para

encontrar as soluções da equação $ax^2 + bx + c = 0$, onde a, b e c são constantes reais e $a \neq 0$. Podemos usar a fórmula resolutiva de Bhaskara dada por

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{3.1}$$

onde a, b e c são constantes. Vamos ver um exemplo de como encontrar as soluções de uma equação quadrática de forma analítica utilizando essa fórmula.

EXEMPLO 3.2.2: Encontre os zeros da função $f(x) = x^2 - 6x + 5$ de forma analítica.

Escrevendo a equação na forma $x^2 - 6x + 5 = 0$, podemos resolvê-la aplicando a fórmula de Bhaskara, conforme apresentado na equação (3.1).

$$x = \frac{-(-6) \pm \sqrt{(-6)^2 - 4 \cdot 1 \cdot 5}}{2 \cdot 1}$$

$$x = \frac{6 \pm \sqrt{36 - 20}}{2}$$

$$x = \frac{6 \pm \sqrt{16}}{2}$$

$$x = \frac{6 \pm 4}{2}.$$

Portanto, as soluções são:

$$x_1 = \frac{6 + 4}{2} = 5$$

$$x_2 = \frac{6 - 4}{2} = 1.$$

A Figura 3.12 apresenta o gráfico da função $f(x) = x^2 - 6x + 5$ destacando seus zeros (código).

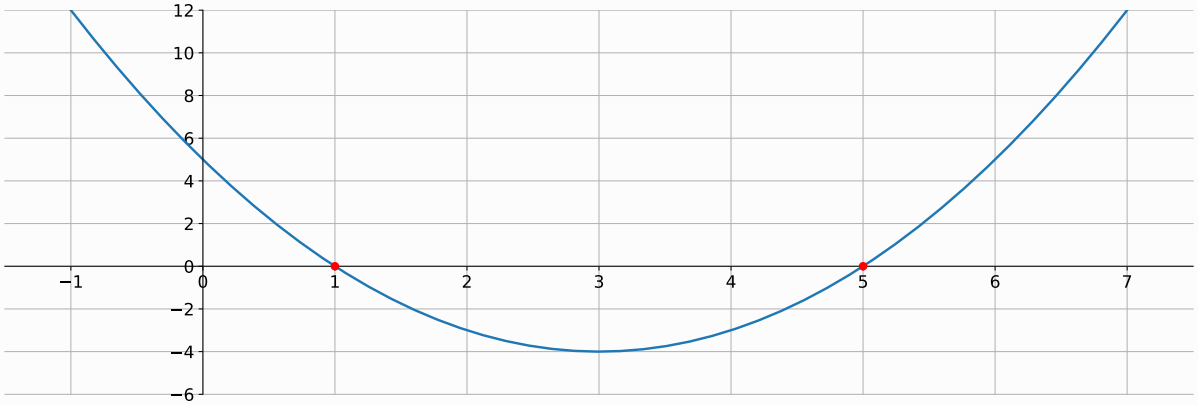

Figura 3.12: Gráfico de $f(x) = x^2 - 5x + 6$ com seus zeros destacados.

Vamos analisar agora a função trigonométrica $f(x) = \text{sen}(x)$. Conhecemos os zeros dessa função com base em suas propriedades (círculo trigonométrico). Os zeros da função seno ocorrem em múltiplos de π, ou seja, $x = -\pi,\ 0,\ \pi,\ 2\pi,\ 3\pi,\ \ldots$.

A Figura 3.13 apresenta o gráfico da função $f(x) = \text{sen}(x)$ destacando seus zeros (código).

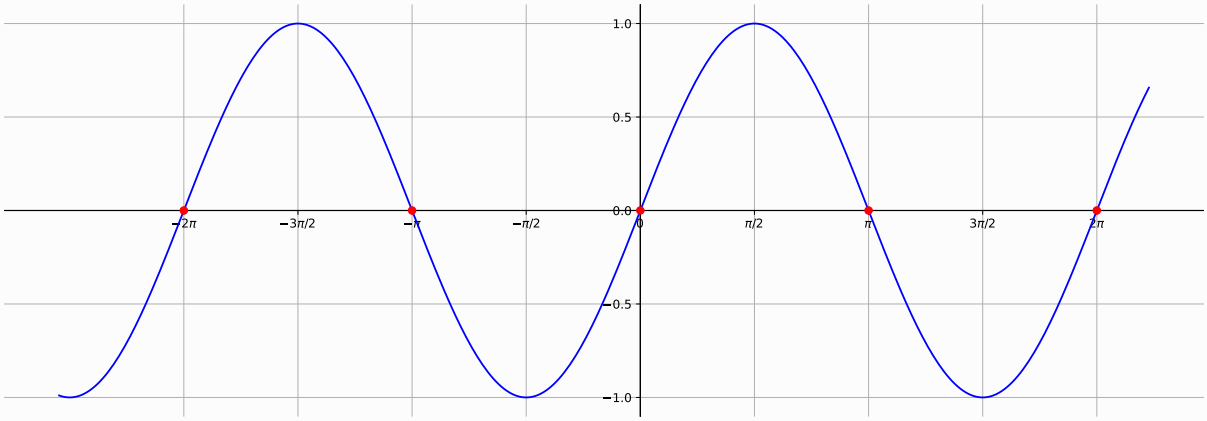

Figura 3.13: Gráfico de $f(x) = \text{sen}(x)$ com seus zeros destacados.

Vamos considerar agora uma função mais complexa, como $f(x) = x - \text{sen}(x)$. Diferente da função seno simples, onde os zeros são múltiplos de π, os zeros de $f(x) = x - \text{sen}(x)$ são difíceis de serem encontrados analiticamente. Exemplos como esse ilustram a importância dos métodos numéricos, que serão explorados no Capítulo 4, métodos esses que permitem aproximar soluções de forma prática para funções complexas onde métodos analíticos não são viáveis.

Ao compreendermos os conceitos fundamentais das funções na matemática, como domínio, contradomínio, imagem, zeros, entre outros, ampliamos nossa capacidade de

resolver problemas de forma eficiente e precisa. Nesta seção, exploramos as funções na matemática, agora, daremos um passo adiante onde vamos explorar as funções em Python. Na próxima seção, faremos uma apresentação breve sobre esse tipo de função utilizando a linguagem Python para criar e manipular funções, abrindo um novo horizonte de possibilidades para resolver problemas matemáticos e computacionais.

3.3 Funções em Python

As funções em Python são blocos de código que realizam tarefas específicas e podem ser reutilizadas em diferentes partes de um programa. Para definir uma função em Python, utilizamos a palavra-chave `def` seguida pelo nome da função e parênteses que podem ou não conter os argumentos da função. Esses argumentos são os valores que a função recebe para executar suas ações. Para utilizar uma função, basta chamá-la pelo nome, seguido pelos parênteses que podem conter os argumentos necessários para a execução da função, caso ela os exija. Essa abordagem ajuda a organizar o código, tornando-o mais legível e reutilizável. Além disso, as funções podem retornar valores, que são os resultados das operações realizadas pela função. Isso permite que o programa receba e manipule esses valores de retorno conforme necessário.

Uma função em Python segue a sintaxe:

```
1 def nome_da_funcao(argumentos):
2     # Corpo da função
3     return resultado
```

Cada elemento do código desempenha um papel específico, vamos explicar o significado de cada um deles a seguir.

- `def`: palavra-chave para definir uma função.
- `nome_da_funcao`: nome que você escolhe para sua função.
- `argumentos`: valores que a função recebe para executar suas ações.
- `return`: palavra-chave para retornar um resultado da função.

Vamos explorar vários exemplos para melhor compreendermos na prática o funcionamento das funções em Python.

Inicialmente vamos ver um exemplo onde escreveremos um código Python que define uma função de soma simples.

EXEMPLO 3.3.1: Escreva um código Python que define uma função de soma simples (código).

A função de soma simples pode ser definida da seguinte forma.

```
1 # Definindo a função
2 def soma(a, b):
3   return a + b
```

No código, a palavra-chave `def` é usada para definir a função `soma(a, b)` que receberá dois argumentos, `a` e `b`, e retornará a soma desses dois valores. A partir de agora podemos reutilizá-la em qualquer outro código, apenas chamando-a pelo nome e fornecendo os valores a serem atribuídos para os argumentos, por exemplo

```
1 # Chamando a função
2 resultado = soma(3, 5)
3 print("A soma de 3 e 5 é:", resultado)
```

Esse código gera a resposta apresentada a seguir.

```
A soma de 3 e 5 é: 8
```

Dentro da função, a instrução `return a + b` especifica que o resultado da soma de `a` e `b` deve ser retornado quando a função é chamada.

Vamos ver um exemplo onde a função é definida e retorna os zeros de uma função quadrática.

EXEMPLO 3.3.2: Escreva um código Python que define uma função para calcular os zeros da função quadrática $f(x) = ax^2 + bx + c$ (código).

Para definirmos uma função em Python que recebe os coeficientes a, b e c de uma função quadrática e retorna os zeros da função, devemos primeiro verificar o discriminante, que nos dirá se a equação possui duas raízes reais e distintas, duas raízes reais e iguais ou não possui raízes reais. Em seguida, calcularemos as raízes, se existirem. Podemos utilizar o código a seguir.

```
1 # Importando a biblioteca math
2 import math
3
4 # Definindo a função
5 def bhaskara(a, b, c):
```

```
    delta = b**2 - 4*a*c
    if delta > 0:
        x1 = (-b + math.sqrt(delta)) / (2*a)
        x2 = (-b - math.sqrt(delta)) / (2*a)
        return x1, x2
    elif delta == 0:
        x = -b / (2*a)
        return x
    else:
        return "Não há raízes reais"
```

O código define uma função chamada `bhaskara` que calcula as raízes de uma equação quadrática $ax^2+bx+c=0$ utilizando a fórmula de Bhaskara (3.1). A função começa importando o módulo `math` para usar a função `sqrt`, que calcula a raiz quadrada. Em seguida, define-se a função `bhaskara` que aceita três parâmetros: `a`, `b` e `c`, que são os coeficientes da equação quadrática. Dentro da função, o discriminante (Δ) é calculado como `b**2 - 4*a*c`. Se o discriminante for positivo (`delta > 0`), a função calcula e retorna as duas raízes reais `x1` e `x2`. Se o discriminante for zero (`delta == 0`), a função calcula e retorna a raiz real única `x`. Se o discriminante for negativo, a função retorna a mensagem `"Não há raízes reais"`.

A partir de agora podemos reutilizá-la em qualquer outro código, apenas chamando-a pelo nome e fornecendo os valores a serem atribuídos para os argumentos. Vamos ver o exemplo a seguir.

EXEMPLO 3.3.3: Escreva um programa em Python que utilize a função `bhaskara` para encontrar os zeros das funções $f(x)=x^2-7x+6$, $g(x)=x^2-4x+4$ e $h(x)=x^2-x+10$ (código).

Para encontrar os zeros das funções quadráticas $f(x)$, $g(x)$ e $h(x)$ utilizando a função `bhaskara` em Python, é necessário primeiro atribuir os valores dos coeficientes a, b e c de cada equação quadrática. Em seguida, chamamos a função `bhaskara` com esses valores para calcular as raízes da equação.

```
# Chamando a função e atribuindo os argumentos

raízes_f = bhaskara(1, -7, 6)
print(f"Os zeros da função f(x) são: {raízes_f}")

raízes_g = bhaskara(1, -4, 4)
```

```
print(f"Os zeros da função g(x) são: {raízes_g}")

raízes_h = bhaskara(1, -1, 10)
print(f"Os zeros da função h(x) são: {raízes_h}")
```

Esse código gera o resultado apresentado a seguir.

```
Os zeros da função f(x) são: (6, 1)
Os zeros da função g(x) são: 2
Os zeros da função h(x) são: Não há raízes reais
```

Este código Python demonstra como utilizar a função `bhaskara` para calcular os zeros das funções quadráticas $f(x)$, $g(x)$ e $h(x)$. Cada chamada da função `bhaskara` é feita com os coeficientes específicos de cada função quadrática, e os resultados são impressos para indicar os zeros de cada função, mostrando também a mensagem adequada quando não há raízes reais para a função como no caso da função $h(x)$.

Ao compreender como definir funções em Python, passar argumentos para elas e receber valores de retorno, os programadores têm uma ferramenta poderosa para organizar e reutilizar código de maneira eficiente. É importante notar a distinção entre funções na matemática e em Python: enquanto as primeiras representam relações entre conjuntos, as segundas são blocos reutilizáveis que encapsulam tarefas específicas.

No próximo capítulo, exploraremos métodos numéricos para encontrar zeros de funções. Estes métodos são fundamentais para resolver equações complexas que não podem ser resolvidas de forma analítica, oferecendo técnicas eficazes e algoritmos poderosos que são amplamente utilizados em diversas áreas da ciência e engenharia.

4

Métodos Numéricos para Zeros de Funções

Os métodos numéricos são técnicas computacionais utilizadas para resolver problemas matemáticos que podem não ter solução analítica direta. Eles são particularmente úteis quando lidamos com equações complexas ou funções onde seus zeros não podem ser encontrados facilmente. Neste trabalho, serão utilizados métodos numéricos iterativos na busca por zeros de funções.

Os métodos numéricos iterativos são uma categoria importante desses métodos, que utilizam estimativas iniciais para as soluções, que são valores aproximados iniciais fornecidos como ponto de partida para o processo iterativo. Estas estimativas iniciais são aprimoradas repetidamente até alcançar uma precisão desejada. Esse aprimoramento é realizado por meio de iterações, que consistem no processo de repetir cálculos para melhorar a precisão da solução. Durante as iterações, os métodos numéricos empregam técnicas específicas para ajustar as estimativas, movendo-as em direção à solução correta. Esse movimento em direção à solução é chamado de convergência, e acontece gradualmente até que a solução atinja um nível aceitável de precisão.

Esses métodos são fundamentais em diversas áreas, como engenharia, física, estatística e ciência da computação, proporcionando uma abordagem prática e eficiente para resolver problemas matemáticos complexos.

4.1 Método para Calcular a Raiz Quadrada

Vamos começar explorando um método para calcular a raiz quadrada de um determinado número. No contexto dos métodos numéricos iterativos, utilizamos um índice n para indicar a posição de cada estimativa na sequência. Por exemplo, x_n representa nossa estimativa atual e x_{n+1} é a próxima estimativa. Essa nova estimativa é então usada na próxima iteração do método. A letra n como índice ajuda a indicar a ordem das estimativas na sequência, facilitando a compreensão de como cada estimativa se baseia na anterior.

Suponha que desejamos encontrar a raiz quadrada de um número a. Inicialmente, fazemos uma estimativa inicial x_0. Em cada iteração, atualizamos nossa estimativa usando a fórmula

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right). \tag{4.1}$$

Essa fórmula é derivada de um método matemático chamado Método de Newton, do qual falaremos com maiores detalhes na Seção 4.2. Ela deve ser aplicada repetidamente até que a diferença entre duas interações consecutivas seja suficientemente pequena para considerarmos a solução como precisa o bastante.

Vamos calcular a raiz quadrada de 9 usando a fórmula (4.1). É claro que sabemos que a resposta é 3, mas vamos usar esse exemplo para entender o funcionamento do método.

Exemplo 4.1.1: Calcular o valor aproximado da raiz quadrada de 9 com 4 casas decimais de precisão.

Vamos calcular a raiz quadrada do número 9 usando como estimativa inicial $x_0 = 4$ e vamos repetir os cálculos com as novas estimativas até obtermos 4 casas decimais precisas.

$$x_0 = 4$$

$$
\begin{aligned}
x_1 &= \frac{1}{2}\left(4 + \frac{9}{4}\right) && = 3{,}125 \\
x_2 &= \frac{1}{2}\left(3{,}125 + \frac{9}{3{,}125}\right) && = 3{,}0025 \\
x_3 &= \frac{1}{2}\left(3{,}0025 + \frac{9}{3{,}0025}\right) && = 3{,}0000
\end{aligned}
$$

Após três iterações, obtivemos a resposta com 4 casas decimais de precisão.

Observe que em cada cálculo realizado, empregamos o valor da estimativa anterior, evidenciando assim a natureza iterativa desse método. Agora, vamos aplicar essa ideia para calcular a raiz quadrada de 7 usando a fórmula (4.1). Como a raiz quadrada de 7 é irracional, podemos ver como a fórmula iterativa é eficiente para aproximar a solução desejada.

EXEMPLO 4.1.2: Calcular o valor aproximado da raiz quadrada de 7 com 4 casas decimais de precisão.

Vamos calcular a raiz quadrada do número 7 usando como estimativa inicial $x_0 = 3$ e vamos repetir os cálculos com as novas estimativas até obtermos 4 casas decimais precisas.

$$
\begin{aligned}
x_0 &= 3 \\
x_1 &= \frac{1}{2}\left(3 + \frac{7}{3}\right) && = 2{,}6666 \\
x_2 &= \frac{1}{2}\left(2{,}6666 + \frac{7}{2{,}6666}\right) && \approx 2{,}6458 \\
x_3 &= \frac{1}{2}\left(2{,}6458 + \frac{7}{2{,}6458}\right) && \approx 2{,}6457 \\
x_4 &= \frac{1}{2}\left(2{,}6457 + \frac{7}{2{,}6457}\right) && \approx 2{,}6457
\end{aligned}
$$

Observe que da terceira para a quarta iteração os algarismos das primeiras 4 casas decimais não foram alterados, logo, após quatro iterações temos uma estimativa da raiz quadrada de 7, sendo $2{,}6457$, com precisão de 4 casas decimais. Podemos verificar que $2{,}6457^2 = 6{,}9997$, ou seja, bem próximo de 7.

Vamos agora utilizar a programação para realizar esses cálculos de forma computacional. Uma das vantagens de usar o computador é a capacidade de realizar muitas

iterações rapidamente, alcançando a precisão desejada de maneira eficiente.

Primeiramente vamos definir a função `raiz_quadrada` que se utilizará da fórmula (4.1). Essa função será responsável para efetuar os cálculos até que se obtenha um resultado satisfatório (código).

```
# Definindo a função raiz quadrada
def raiz_quadrada(a, x0):
    print(f'Estimativa inicial: x0 = {x0}')
    estimativa_x = x0

    # Realiza até 100 iterações para calcular a raiz quadrada
    for iteracao in range(1, 100):
        x0 = 0.5 * (x0 + a / x0)
        print(f'Iteração {iteracao}: x{iteracao} = {x0}')

        # Verifica se a aproximação é sufuciente
        if abs(estimativa_x - x0) < 1e-4:
            break
        estimativa_x = x0

    # Retorna a estimativa final da raiz quadrada
    return x0
```

A função `raiz_quadrada` recebe dois argumentos: `a`, que é o número para o qual se deseja encontrar a raiz quadrada, e `x0`, a estimativa inicial. A função começa imprimindo a estimativa inicial. Em seguida, o laço `for` realiza até 100 iterações para calcular a nova estimativa da raiz quadrada utilizando a fórmula iterativa `x0 = 0.5 * (x0 + a / x0)`. A condição de parada verifica, utilizando `if`, se a diferença absoluta entre `estimativa_x` e `x0` é menor que `1e-4`, o que equivale a 0,0001, ou seja, 4 casas decimais de precisão. Se essa condição for atendida, a sequência é interrompida utilizando `break`. A função retorna a estimativa final da raiz quadrada.

Agora vamos escrever um código que se utiliza da função `raiz_quadrada` para calcular a raiz quadrada de um número fornecido pelo usuário.

> Exemplo 4.1.1: Escreva um código Python que solicita ao usuário um número e uma estimativa inicial, e utiliza a função `raiz_quadrada` para calcular a raiz quadrada desse número (código).

Ao iniciar o programa, é solicitado ao usuário digitar o número para encontrar a raiz quadrada e posteriormente digitar a estimativa inicial. Após, o programa mostra a estimativa inicial e gera todos os resultados encontrados de todas as iterações até que se obtenha a precisão desejada.

```
# Solicita um número do usuário e a estimativa inicial
a = float(input('Digite o número para encontrar a raiz
    quadrada: '))
x0 = float(input('Digite a estimativa inicial: '))

# Chama a função raiz_quadrada e executa o método
resultado = raiz_quadrada(a, x0)

# Exibe o resultado
print(f'\nA raiz quadrada de {a} é aproximadamente
    {resultado}')
```

O código solicita ao usuário que insira um número e uma estimativa inicial usando o comando `input`. Esses valores são armazenados nas variáveis `a` e `x0`, respectivamente. A função `raiz_quadrada` é então chamada com esses argumentos, realizando o cálculo da raiz quadrada. O resultado é armazenado na variável `resultado` e exibido ao usuário com o comando `print`, mostrando a raiz quadrada aproximada do número fornecido.

Supondo que o usuário tenha digitado o número 17 para se calcular a raiz quadrada e tenha digitado o número 8 para a estimativa inicial, o programa gera a reposta a seguir.

```
Digite o número para encontrar a raiz quadrada: 17
Digite a estimativa inicial: 8
Estimativa inicial: x0 = 8,0
Iteração 1: x1 = 5,0625
Iteração 2: x2 = 4,2103
Iteração 3: x3 = 4,1240
Iteração 4: x4 = 4,1231
Iteração 5: x5 = 4,1231

A raiz quadrada de 17,0 é aproximadamente 4,1231.
```

Observe que o programa interrompe as iterações quando os 4 primeiros dígitos após a vírgula não se alteraram, tendo assim uma precisão de 4 casas decimais. Podemos

verificar que $4{,}1231^2 = 16{,}9999$, ou seja, bem próximo de 17.

Podemos também utilizar esse programa para o cálculo da raiz quadrada de números decimais. Supondo que o usuário tenha digitado o número decimal 19,4 para se calcular a raiz quadrada e tenha digitado o número 5,8 para a estimativa inicial, o programa gera a reposta a seguir.

```
Digite o número para encontrar a raiz quadrada: 19.4
Digite a estimativa inicial: 5.8
Estimativa inicial: x0 = 5.8
Iteração 1: x1 = 4.5724
Iteração 2: x2 = 4.4076
Iteração 3: x3 = 4.4045
Iteração 4: x4 = 4.4045

A raiz quadrada de 19.4 é aproximadamente 4.4045.
```

O programa obteve 4 casas decimais de precisão. Podemos verificar que $4{,}4045^2 = 19{,}3996$, ou seja, bem próximo de $19{,}4$.

A programação de cálculos iterativos oferece várias vantagens, incluindo rapidez, precisão e automação, evitando erros manuais e facilitando a análise de problemas complexos. O código apresentado, que calcula a raiz quadrada com precisão de quatro casas decimais, demonstra a eficácia desse método computacional. Executar o código no Google Colab realça sua praticidade, pois elimina a necessidade de instalações adicionais, tornando a programação mais acessível para todos.

Encerramos esta seção com uma pergunta instigante: como calcular a raiz cúbica de um número ou qualquer outra raiz? Essa questão será respondida na próxima seção, onde exploraremos o Método de Newton como uma ferramenta poderosa para lidar não somente com o cálculo de raízes de números reais, mas também para solucionar uma vasta quantidade de problemas matemáticos.

4.2 Método de Newton

Como podemos resolver outras raízes e equações além da raiz quadrada? Essa é uma pergunta natural que surge quando exploramos métodos iterativos como a fórmula (4.1). Podemos resolver diversas raízes e equações utilizando fórmulas prontas. Aqui estão algumas delas.

Raiz Quadrada: Para encontrar a raiz quadrada de um número a, podemos usar a fórmula

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

Raiz Cúbica: A fórmula para a raiz cúbica de um número a é

$$x_{n+1} = \frac{1}{3}\left(2x_n + \frac{a}{x_n^2}\right).$$

Raiz n-ésima: A fórmula geral para a raiz n-ésima de um número a é

$$x_{n+1} = \frac{1}{n}\left((n-1)x_n + \frac{a}{x_n^{n-1}}\right).$$

Essas fórmulas foram geradas através do Método de Newton e nos permitem encontrar raízes de maneira eficiente. O Método de Newton foi desenvolvido inicialmente pelo grande Sir Isaac Newton, um dos maiores cientistas de todos os tempos. Se trata de um procedimento matemático iterativo usado para encontrar aproximações para um zero de uma função. Começamos com uma estimativa inicial e, em cada iteração, usamos a derivada da função para calcular uma nova estimativa mais próxima do zero desejado. A fórmula geral do Método de Newton para encontrar um zero de uma função $f(x)$ é dada por

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

onde $f'(x)$ é a derivada de $f(x)$. No caso das raízes n-ésimas, aplicamos essa fórmula específica à função apropriada para obter as fórmulas mencionadas anteriormente.

No entanto, esse conhecimento matemático envolve cálculos mais avançados, normalmente estudados no curso superior. Para nosso propósito, é suficiente utilizar as

fórmulas prontas sem a necessidade de compreender detalhadamente o processo de como foram derivadas.

Podemos expressar o cálculo de raízes como sendo zeros de funções específicas, por exemplo, a raiz quadrada de a pode ser representada como o zero da função $f(x) = x^2 - a$. Quando encontramos o valor de x que torna $f(x) = 0$, obtemos a raiz quadrada de a, ou seja,

$$\begin{aligned} x^2 - a &= 0 \\ x^2 &= a \\ |x| &= \sqrt{a} \\ x &= \pm\sqrt{a} \end{aligned}$$

Da mesma forma, podemos encontrar a raiz cúbica de um número a como o zero da função $g(x) = x^3 - a$. Para encontrar a raiz cúbica, procuramos o valor de x que zera $g(x)$, ou seja,

$$\begin{aligned} x^3 - a &= 0 \\ x^3 &= a \\ x &= \sqrt[3]{a} \end{aligned}$$

Além disso, outras funções podem ser tratadas da mesma maneira. Por exemplo, para encontrar o zero da função trigonométrica $f(x) = \text{sen}(x)$, usamos o Método de Newton para encontrar o valor de x que faz $\text{sen}(x) = 0$. O Método de Newton pode ser aplicado a uma variedade de funções, incluindo polinômios, funções trigonométricas, exponenciais, e muitas outras.

Geometricamente, o Método de Newton pode ser interpretado como o processo de encontrar a interseção da reta tangente à curva da função em um ponto dado inicialmente com o eixo x. A cada iteração, a reta tangente é recalculada com base na inclinação da curva no ponto atual, aproximando-se cada vez mais do zero da função.

A Figura 4.1 mostra o gráfico da função $f(x) = x^2 - 4$ e as retas tangentes geradas pelo Método de Newton. Iniciando pela estimativa inicial $x_0 = 4$, a cada iteração, a interceção da reta tangente com o eixo x se aproxima cada vez mais do zero da função (código).

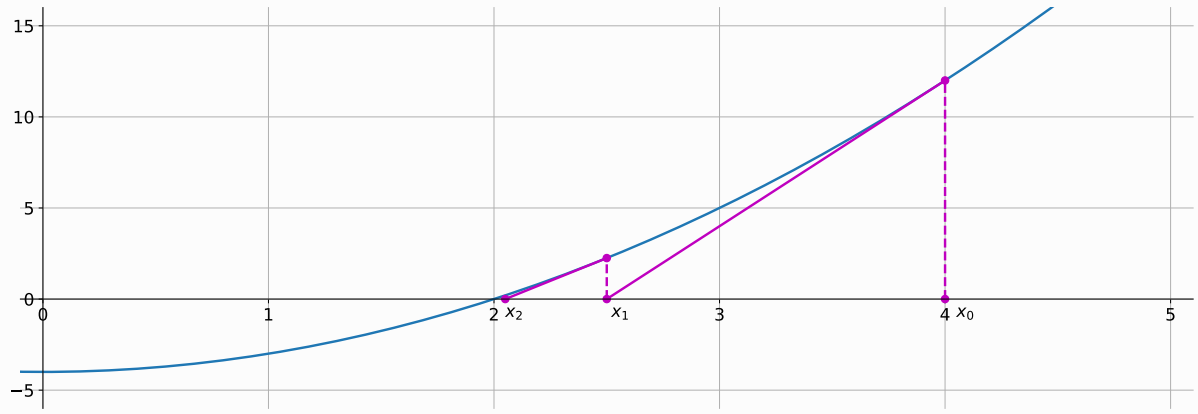

Figura 4.1: Interpretação geométrica do Método de Newton.

Podemos assim utilizar o Método de Newton para funções mais complexas. Vamos fazer um exemplo com uma função trigonométrica.

EXEMPLO 4.2.1: Escreva um código Python que encontre um zero da função $f(x) = \text{sen}(x)$ utilizando o Método de Newton (código).

Sabemos que o $\text{sen}(\pi) = 0$, logo as iterações do Método de Newton vão aproximar o valor da estimativa inicial ao valor de π ou à algum múltiplo de π. A fórmula de iteração específica para esse cálculo será

$$x_{n+1} = x_n - \frac{\text{sen}(x_n)}{\cos(x_n)}.$$

Apresentaremos a seguir o código que encontra um zero da função $f(x) = \text{sen}(x)$ através de uma estimativa inicial suficientemente próxima utilizando o Método de Newton.

```
1  # Importa a biblioteca math
2  import math
3
4  #Define o Método de Newton
5  def metodo_newton(aprox_inicial, precisao):
6      x = aprox_inicial
7      iteracao = 0
8      while True:
9          iteracao += 1
10         x_novo = x - math.sin(x) / math.cos(x)
11         print(f"Iteração {iteracao}: x_{iteracao} = {x_novo}")
12         if abs(x_novo - x) < precisao:
```

```
            return x_novo
        x = x_novo

# Solicita ao usuário as informações iniciais
ap_inicial = float(input('Digite a estimativa inicial: '))
precisao = float(input('Digite a precisão desejada: '))

# Executa o método e imprime os resultados
resultado = metodo_newton(ap_inicial, precisao)
print(f'\nO resultado final aproximado é {resultado}')
```

Ao executar este código, informe a estimativa inicial e a precisão. Se o valor inicial estiver suficientemente próximo de um dos zeros da função seno, o programa gerará uma sequência que se aproxima cada vez mais do zero da função.

No código, o usuário fornece uma estimativa inicial utilizando a função `input`, que é armazenada na variável `ap_inicial`, e a precisão desejada, armazenada na variável `precisao`. A função `metodo_newton` utiliza esses valores para iterar até que a diferença entre a estimativa atual e a nova seja menor que a precisão. A cada iteração, a variável `x` é atualizada com a fórmula `x_novo = x - math.sin(x) / math.cos(x)` e o progresso é impresso. Quando a condição de precisão é satisfeita, a função retorna `x_novo` como a aproximação do zero da função, que é exibida com a função `print`.

Supondo que o usuário tenha digitado o número 2 para a estimativa inicial e 0,0001 para a precisão desejada, o programa gera o seguinte resultado.

```
Digite a estimativa inicial: 2
Digite a precisão desejada: 0.0001
Iteração 1: x_1 = 4.1850
Iteração 2: x_2 = 2.4679
Iteração 3: x_3 = 3.2662
Iteração 4: x_4 = 3.1409
Iteração 5: x_5 = 3.1416
Iteração 6: x_6 = 3.1416

O resultado final aproximado é 3.1416.
```

Observe que com essa estimativa inicial foram necessárias 6 iterações para obter a aproximação de 4 casas decimais, se a estimativa inicial for mais próxima da solução, menos iterações são necessárias para atingir o objetivo.

Agora se o usuário digitar o número 6 para a estimativa inicial e 0,0001 para a precisão desejada, o programa gera o seguinte resultado.

```
1 Digite a estimativa inicial: 6
2 Digite a precisão desejada: 0.0001
3 Iteração 1: x_1 = 6.2910
4 Iteração 2: x_2 = 6.2831
5 Iteração 3: x_3 = 6.2831
6
7 O resultado final aproximado é 6.2831.
```

Observe que, se o usuário digitar o número 6, o programa se aproximaria da resposta 6,2831, que é um outro zero da função seno(aproximadamente 2π). Ou seja, o Método de Newton se aproxima do zero mais próximo da estimativa inicial.

A Figura 4.2 mostra geometricamente como três iterações do Método de Newton geraram tangentes em que o ponto de interceção com o eixo x se aproximam graduadamente do zero da função (código).

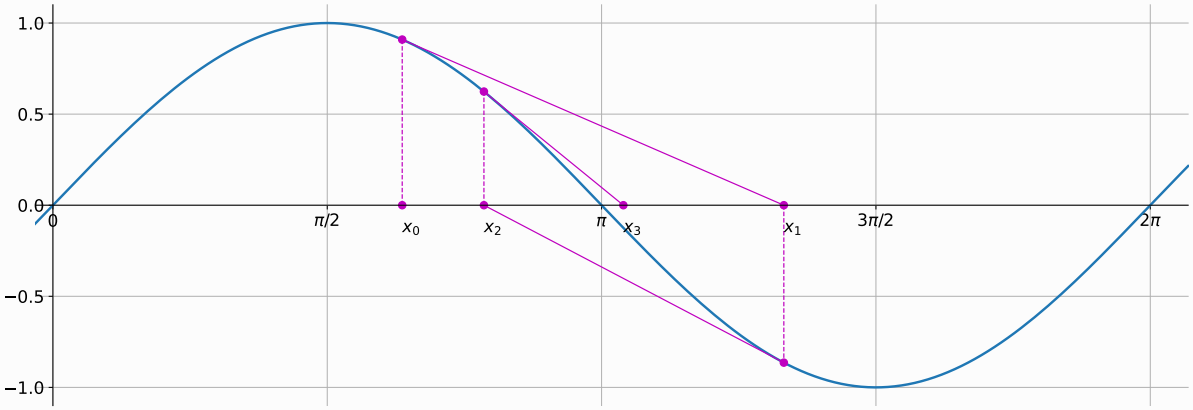

Figura 4.2: Tangentes geradas pelo Método de Newton.

Em alguns casos, dependendo da estimativa inicial, o Método de Newton não converge para o zero da função. Por exemplo, se a estimativa inicial for $\frac{\pi}{2}$, a reta tangente gerada pelo Método de Newton não será direcionada para o zero da função.

A Figura 4.3 mostra geometricamente como a reta tangente não vai em direção ao zero da função (código).

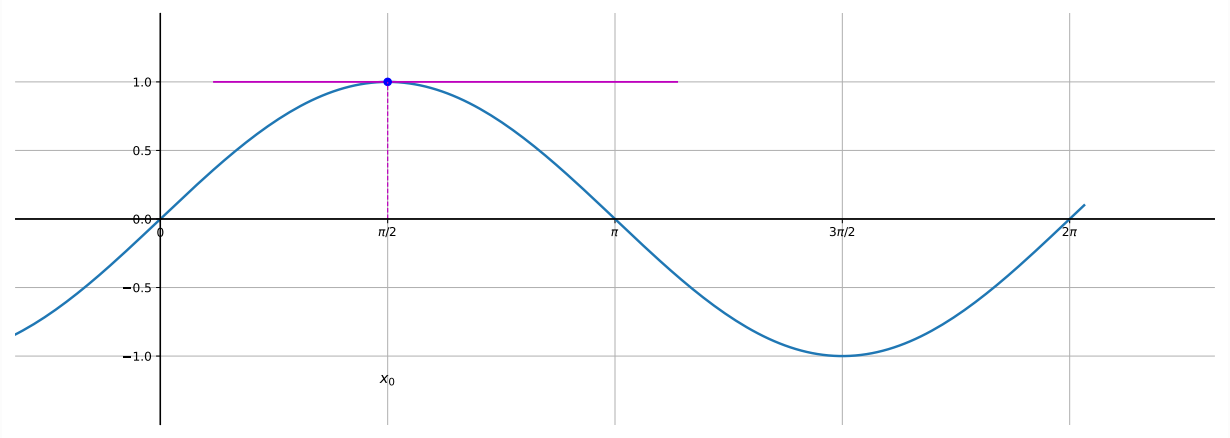

Figura 4.3: Caso em que o Método de Newton não converge.

É importante que a estimativa inicial esteja suficientemente próxima da raiz para que o Método de Newton convirja de forma eficiente. Quando não temos informações suficientes sobre a função para escolher uma estimativa adequada, isso pode ser uma desvantagem significativa do Método de Newton, levando à divergência ou à convergência para uma raiz diferente da desejada.

Aplicar o Método de Newton de forma eficaz exige compreender como calcular a inclinação do gráfico em um ponto, relacionado ao conceito de derivada do cálculo avançado, o que está fora do escopo desta apostila. Nas próximas seções, apresentaremos o Método da Bisseção e o Método da Secante, que, embora exijam mais iterações, não necessitam da derivada. Esses métodos são alternativas para encontrar zeros de funções quando a derivada não é facilmente calculável ou se deseja evitar técnicas matemáticas mais avançadas.

4.3 Método da Bisseção

O Método da Bisseção é uma técnica simples e eficaz para encontrar um zero de uma função em um intervalo conhecido. Ele se baseia no fato de que, se a função é contínua, ou seja, não possui interrupções, buracos ou saltos no gráfico, e muda de sinal entre os extremos do intervalo, deve haver pelo menos um zero dentro desse intervalo. O método utiliza essa propriedade das funções contínuas para localizar sistematicamente o zero da função no intervalo fornecido.

A ideia básica do método é dividir o intervalo pela metade repetidamente até que se obtenha uma aproximação satisfatória do zero da função. A fórmula de iteração do

Método da Bisseção é dada por

$$x_{n+1} = \frac{a_n + b_n}{2}, \tag{4.2}$$

onde x_{n+1} é o próximo ponto intermediário, a_n e b_n são os extremos do intervalo anterior. Essa técnica aproveita a mudança de sinal da função nos extremos do intervalo para garantir que o zero da função está contido no novo intervalo resultante da bisseção.

Ao realizar várias iterações do Método da Bisseção, podemos aproximar o zero da função com a precisão desejada, sem a necessidade de calcular derivadas ou outros conceitos avançados de análise matemática.

Vamos exemplificar o método buscando a raiz quadrada de 9, que é equivalente a encontrar o zero da função $f(x) = x^2 - 9$. Embora já saibamos que a raiz quadrada de 9 é 3, utilizaremos este exemplo para demonstrar como o Método da Bisseção funciona. Consideramos o intervalo entre 1 e 4, onde sabemos que a raiz está contida. Em cada iteração, ajustamos os extremos do intervalo para garantir que o zero da função permaneça dentro dele, ou seja, um extremo deve ser menor que 3 e o outro maior que 3. Assim, o Método da Bisseção permite refinar a estimativa da raiz de forma sistemática.

EXEMPLO 4.3.1: Encontre um valor aproximado para um zero da função $f(x) = x^2 - 9$, utilizando o Método da Bisseção no intervalo entre 1 e 4.

Vamos buscar o zero da função $f(x) = x^2 - 9$, que se trata de buscar o valor da raiz quadrada de 9 no intervalo entre 1 e 4. Para isso, avaliamos $f(1)$ e $f(4)$:

$$f(1) = 1^2 - 9 = -8 \quad \text{e} \quad f(4) = 4^2 - 9 = 7.$$

Como $f(1) < 0$ e $f(4) > 0$, sabemos que existe uma raiz no intervalo entre 1 e 4.

A primeira iteração consiste em calcular o ponto médio

$$x_1 = \frac{1+4}{2} = 2{,}5.$$

Agora avaliamos $f(2{,}5)$.

$$f(2{,}5) = 2{,}5^2 - 9 = -2{,}75.$$

Como $f(2{,}5) < 0$ e $f(4) > 0$, o novo intervalo será $[2{,}5; 4]$.

Na segunda iteração, calculamos o novo ponto médio

$$x_2 = \frac{2{,}5 + 4}{2} = 3{,}25.$$

Avaliaremos $f(3{,}25)$:

$$f(3{,}25) = 3{,}25^2 - 9 = 1{,}562.$$

Como $f(2{,}5) < 0$ e $f(3{,}25) > 0$, o novo intervalo será $[2{,}5; 3{,}25]$.

E assim sucessivamente, até encontrarmos um valor que esteja dentro da aproximação desejada para o zero da função. As próximas iterações mostram os seguintes resultados.

$$\begin{aligned}
x_3 &= \frac{2{,}5 + 3{,}25}{2} &&= 2{,}875 \\
x_4 &= \frac{2{,}875 + 3{,}25}{2} &&= 3{,}0625 \\
x_5 &= \frac{2{,}875 + 3{,}0625}{2} &&= 2{,}96875 \\
x_6 &= \frac{2{,}96875 + 3{,}0625}{2} &&= 3{,}0156 \\
x_7 &= \frac{2{,}96875 + 3{,}0156}{2} &&= 2{,}9921 \\
x_8 &= \frac{2{,}9921 + 3{,}0156}{2} &&= 3{,}0039.
\end{aligned}$$

Podemos observar que após 8 iterações conseguimos a aproximação para a raiz quadrada de 9 com duas casas decimais de precisão.

Geometricamente o Método da Bisseção pode ser apresentado pelas Figuras 4.4, 4.5, 4.6 e 4.7, apresentando quatro iterações no gráfico da função $f(x) = x^2 - 9$ com um zero em $x = 3$ (código).

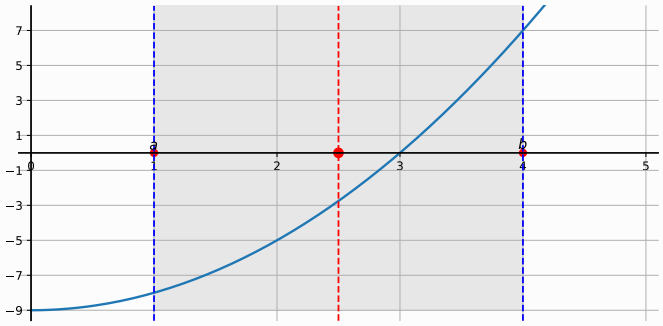

Figura 4.4: Iteração 1: $x_1 = 2{,}5$.

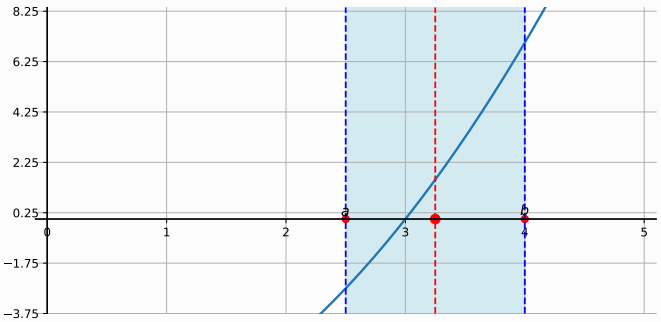

Figura 4.5: Iteração 2: $x_2 = 3{,}25$.

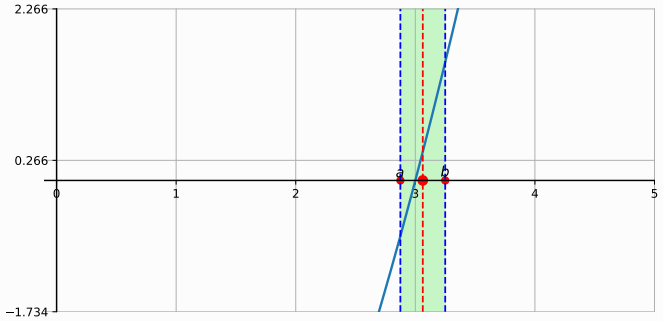

Figura 4.6: Iteração 3: $x_2 = 2{,}875$.

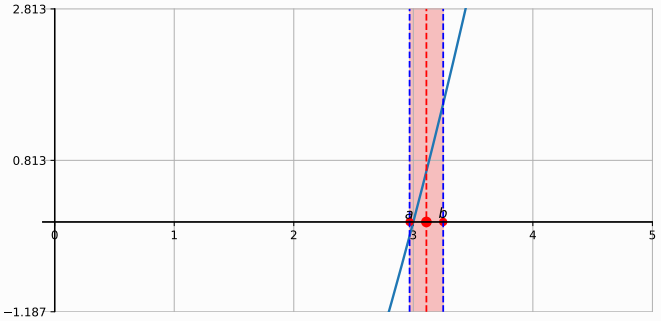

Figura 4.7: Iteração 4: $x_2 = 3{,}065$.

Podemos observar que os intervalos estão diminuindo em torno do zero da função, ilustrando claramente a convergência do método.

Vamos agora mostrar um exemplo com um grau de dificuldade maior, vamos em busca do zero da função exponencial $f(x) = 2^x - 3$.

Podemos verificar que $f(1) = 1^2 - 9 = -8$ e $f(4) = 4^2 - 9 = 7$. Como a função é contínua e há uma mudança de sinal nos extremos do intervalo, existe pelo menos um zero da função nesse intervalo. Este exemplo seria mais difícil com o Método de Newton, pois requer o cálculo da derivada, que envolve conceitos avançados. Portanto, usaremos o Método da Bisseção e aplicaremos um código especialmente desenvolvido para resolver esse exemplo (código).

```
1 # Definindo a função
2 def f(x):
3     return 2**x - 3
4
5 # Definindo o Método da Bisseção
6 def bissecao(a, b, num_iter):
7     for i in range(num_iter):
```

```
        c = (a + b) / 2
        print(f"Iteração {i + 1}: Valor de c = {c}")

        if f(c) == 0:
            return c
        elif f(a) * f(c) < 0:
            b = c
        else:
            a = c

    return (a + b) / 2
# Intervalos iniciais e as iterações
a = 1
b = 2
iteracoes = 8

# Chama a função Bisseção e imprime os resultados
zero = bissecao(a, b, iteracoes)
print(f"O zero da função é aproximadamente {zero}")
```

No código, definimos a função $f(x) = 2^x - 3$ e a função `bissecao(a, b, iter)`, que recebe os extremos do intervalo `a` e `b`, e o número de iterações. Usamos um *loop* `for` para iterar o Método da Bisseção. Calculamos o ponto médio `c` do intervalo e verificamos `f(c)`. Se `f(c) == 0`, retornamos `c`. Caso contrário, atualizamos os extremos do intervalo conforme `f(a)*f(c)`. Após as iterações, retornamos o ponto médio final como a aproximação do zero da função. No exemplo, `a=1`, `b=2`, e são realizadas 8 iterações. O resultado final é impresso como `O zero da função é aproximadamente {zero}`.

Ao executar esse código, após 8 iterações obtemos a seguinte resposta do programa.

```
Iteração 1: Valor de c = 1.50000
Iteração 2: Valor de c = 1.75000
Iteração 3: Valor de c = 1.62500
Iteração 4: Valor de c = 1.56250
Iteração 5: Valor de c = 1.59375
Iteração 6: Valor de c = 1.57812
Iteração 7: Valor de c = 1.58594
Iteração 8: Valor de c = 1.58203
O zero da função é aproximadamente 1.58398
```

Mostrando assim que uma aproximação para um zero da função será 1,58398. Podemos verificar isso substituindo esse valor na função e verificando se o resultado

se aproxima de zero. Temos então que $2^{1,58398} - 3 = -0{,}002$ que é bem próximo de zero.

O Método da Bisseção é conhecido por sua simplicidade e acessibilidade na busca por zeros de funções dentro de um intervalo específico, embora demande mais iterações para alcançar a precisão desejada em comparação a outros métodos. Por outro lado, o Método de Newton oferece uma convergência mais rápida, geralmente exigindo menos iterações, mas requer o cálculo da derivada da função, o que pode ser um obstáculo para quem não está familiarizado com conceitos matemáticos avançados.

Na próxima seção, apresentaremos o Método da Secante como uma alternativa ao Método de Newton. Embora a Secante precise de mais iterações que o Método de Newton, ela é mais rápida que a Bisseção e eficaz em diversas análises numéricas, especialmente quando o cálculo da derivada não é prático.

4.4 Método da Secante

Nesta seção, apresentaremos o Método da Secante, uma técnica iterativa para encontrar zeros de funções sem precisar calcular a derivada. Diferente do Método de Newton, a Secante aproxima a derivada usando dois pontos próximos, tornando-o mais acessível para obter soluções aproximadas dentro de um intervalo.

A fórmula de iteração do Método da Secante é dada por

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}, \tag{4.3}$$

onde x_{n+1} é a próxima estimativa da solução, x_n e x_{n-1} são as estimativas atual e anterior, $f(x_n)$ e $f(x_{n-1})$ são os valores da função nos pontos x_n e x_{n-1}.

Vamos usar o Método da Secante para encontrar um zero da função $f(x) = x^2 - 9$ com estimativas iniciais de $x_0 = 1$ e $x_1 = 4$. Se trata do problema de encontrar a raiz quadrada de 9 no intervalo entre 1 e 4, esse problema é solucionado com os métodos de Newton e da Bisseção, porém vamos começar com esse exemplo para apresentar o Método da Secante e também nos servirá como forma de comparação dos métodos.

EXEMPLO 4.4.1: Encontre um valor aproximado para um zero da função $f(x) = x^2 - 9$, utilizando o Método da Secante.

Vamos calcular um zero da função $f(x) = x^2 - 9$ utilizando o Método da Secante com estimativas iniciais $x_0 = 1$ e $x_1 = 4$. As iterações serão

$$x_2 = 4 - \frac{(4^2 - 9) \cdot (4 - 1)}{(4^2 - 9) - (1^2 - 9)} = 2{,}6$$

$$x_3 = 4 - \frac{(4^2 - 9) \cdot (4 - 2{,}6)}{(4^2 - 9) - (2{,}6^2 - 9)} = 2{,}9393$$

$$x_4 = 4 - \frac{(4^2 - 9) \cdot (4 - 2{,}9394)}{(4^2 - 9) - (2{,}9394^2 - 9)} = 2{,}9912$$

$$x_5 = 4 - \frac{(4^2 - 9) \cdot (4 - 2{,}9912)}{(4^2 - 9) - (2{,}9912^2 - 9)} = 2{,}9987$$

$$x_6 = 4 - \frac{(4^2 - 9) \cdot (4 - 2{,}9987)}{(4^2 - 9) - (2{,}9987^2 - 9)} = 2{,}9998.$$

Após 5 iterações, obtemos uma estimativa $2{,}9998$ para a raiz quadrada de 9, ou seja, bem próximo de 3.

Vamos agora encontrar uma aproximação para um zero da função $f(x) = 3^{x/3} - 4$. Vamos então usar o Método da Secante, mas dessa vez usaremos a programação para isso. O código a seguir é um código escrito especialmente para resolver esse exemplo usando o Método da Secante no intervalo entre 1 e 7 (código).

```
# Define a função f(x)
def f(x):
    return 3**(x/3) - 4

# Implementa o Método da Secante
def secante(x0, x1, iteracoes):
    for i in range(iteracoes):
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        print(f" x{i+2} = {x2}")
        x0 = x1
        x1 = x2
    return x2

# Configuração inicial e execução do método
x0 = 1
x1 = 7
```

```
17 iteracoes = 7
18
19 resultado = secante(x0, x1, iteracoes)
20
21 # Imprime o resultado final
22 print(f"O zero da função é aproximadamente {resultado}")
```

Neste código, definimos a função `f(x)` cujo zero desejamos encontrar, $f(x) = 3^{x/3} - 4$. A função `secante(x0, x1, iteracoes)` executa o Método da Secante com duas estimativas iniciais `x0` e `x1` e um número especificado de iterações. Dentro do *loop* `for`, calculamos as aproximações do zero usando a fórmula do método e atualizamos `x0` e `x1` a cada iteração. O programa inicia com `x0=1` e `x1=7` e realiza 7 iterações, retornando e imprimindo a aproximação do zero como `O zero da função é aproximadamente {resultado}`.

Com esses dados, o programa gera o resultado a seguir.

```
1 x2 = 2.33008
2 x3 = 3.05592
3 x4 = 4.00844
4 x5 = 3.75494
5 x6 = 3.78434
6 x7 = 3.78559
7 x8 = 3.78558
8 O zero da função é aproximadamente 3.78558
```

Observe que a partir da sétima iteração, as quatro primeiras casas decimais não se alteram, ou seja, temos uma aproximação com quatro casas decimais de precisão. Geometricamente, a Figura 4.8 mostra como as interseções das secantes com o eixo x se aproximam gradualmente do zero da função (código).

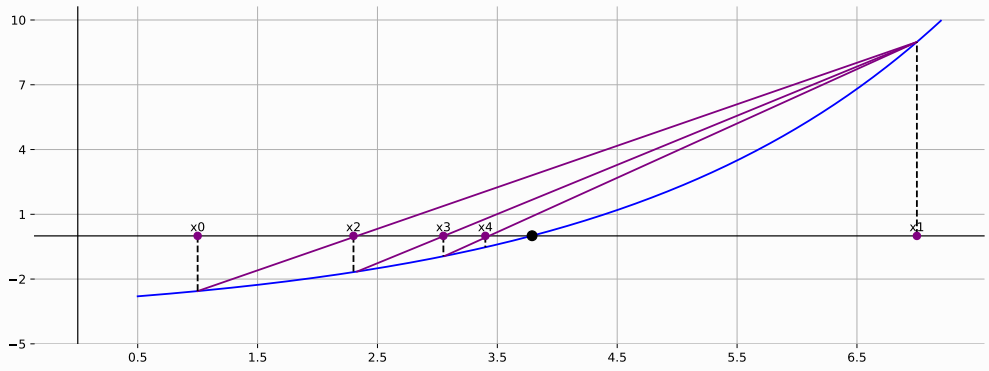

Figura 4.8: Interpretação geométrica do Método da Secante.

Ao comparar os métodos de Newton, Bisseção e Secante, observamos características distintas:

- **Método de Newton**: Converge rapidamente e geralmente requer menos iterações para encontrar uma solução. No entanto, exige o cálculo da derivada, o que pode ser um obstáculo para quem não está familiarizado com cálculo avançado. É importante notar que o Método de Newton pode não convergir se a escolha inicial estiver longe da solução ou se a função não atender às condições necessárias.
- **Método da Bisseção**: Simples e robusto, não necessita da derivada e é aplicável a uma ampla gama de funções. Se a função for contínua no intervalo considerado, o Método da Bisseção garante a convergência, embora possa exigir muitas iterações para alcançar a precisão desejada.
- **Método da Secante**: Oferece uma convergência mais rápida do que a Bisseção, sem precisar calcular a derivada, tornando-o uma alternativa intermediária em termos de velocidade e complexidade. No entanto, o Método da Secante também pode não convergir em alguns casos, dependendo da escolha inicial e das características da função.

Esses métodos possuem suas vantagens e desvantagens, tornando-os mais ou menos adequados dependendo das necessidades específicas do problema.

No próximo capítulo, abordaremos métodos numéricos iterativos para sistemas de equações lineares e não lineares. Esses métodos são essenciais para resolver sistemas complexos encontrados na prática, envolvendo múltiplas variáveis e relações não lineares. Eles são fundamentais em matemática, ciência e engenharia.

5

Métodos Numéricos para Sistemas de Equações

Sistemas de equações algébricas desempenham um papel importante em várias áreas do conhecimento, oferecendo uma poderosa ferramenta para modelar e resolver uma ampla variedade de problemas do mundo real. Neste capítulo, exploraremos duas categorias fundamentais de sistemas de equações: os sistemas lineares e os sistemas não lineares. Compreender a distinção entre essas duas categorias é importante, uma vez que a natureza das equações e os métodos de resolução variam significativamente.

Sistemas de equações lineares envolvem equações com várias variáveis, onde cada variável é multiplicada por uma constante. Para resolver esses sistemas, o número de equações deve ser igual ao número de variáveis, permitindo encontrar os valores que satisfazem todas as equações ao mesmo tempo. No caso de duas variáveis, cada equação representa uma reta, e a solução do sistema é o ponto onde essas retas se cruzam. Com três variáveis, as equações representam planos, e a solução é o ponto onde esses planos se encontram.

Vamos abordar a seguir algumas propriedades e métodos de resolução de sistemas de equações, lineares e não lineares. Ao entender com mais detalhes cada categoria, teremos mais ferramentas e abordagens para resolver uma variedade de problemas complexos que envolvem tais sistemas.

5.1 Sistemas de Equações Lineares

Os sistemas de equações lineares consistem em um conjunto de equações lineares que envolvem as mesmas variáveis. Cada equação no sistema é uma combinação linear dessas variáveis, por exemplo, uma equação com três variáveis pode ser expressa na forma $a_1x + a_2y + a_3z = b$, onde a_1, a_2, a_3 e b são coeficientes constantes. A resolução de um sistema de equações lineares busca encontrar os valores das variáveis que satisfazem simultaneamente todas as equações no sistema. A classificação desses sistemas é baseada no número de soluções que eles admitem. Essa classificação é importante para entender a natureza das soluções e como elas podem ser encontradas.

Sistema Possível: Um sistema possível é aquele que possui pelo menos uma solução, ou seja, existe um conjunto de valores para as variáveis que satisfaz todas as equações do sistema simultaneamente. Este tipo de sistema pode ser classificado de duas formas: determinado e indeterminado. Um sistema é determinado quando possui exatamente uma solução única, o que significa que graficamente as retas que representam as equações se intersectam em um único ponto. Por outro lado, um sistema é indeterminado quando possui infinitas soluções, e graficamente as retas coincidem, indicando que todos os pontos ao longo da reta são soluções do sistema.

Sistema Impossível: Um sistema impossível é aquele que não possui solução. Isso ocorre quando não existe um conjunto de valores para as variáveis que satisfaça todas as equações do sistema simultaneamente, o que acontece graficamente quando as retas são paralelas e não se intersectam.

Vamos analisar três exemplos de sistemas de equações lineares, cada um representando um caso diferente, e explorar sua classificação e representação geométrica no plano cartesiano. Para simplificar a apresentação, focaremos em sistemas 2×2, ou seja, sistemas com duas equações e duas variáveis.

O primeiro exemplo abordará um sistema possível e determinado, ilustrando sua representação no plano cartesiano.

EXEMPLO 5.1.1: Classifique o sistema linear abaixo através da análise da sua representação geométrica no plano cartesiano.

$$\begin{cases} x + y = 6 \\ x - y = 2 \end{cases} \tag{5.1}$$

A Figura 5.1 apresenta geometricamente esse sistema, mostrando as duas retas geradas pelas equações do sistema, bem como o único ponto de interseção entre elas (código).

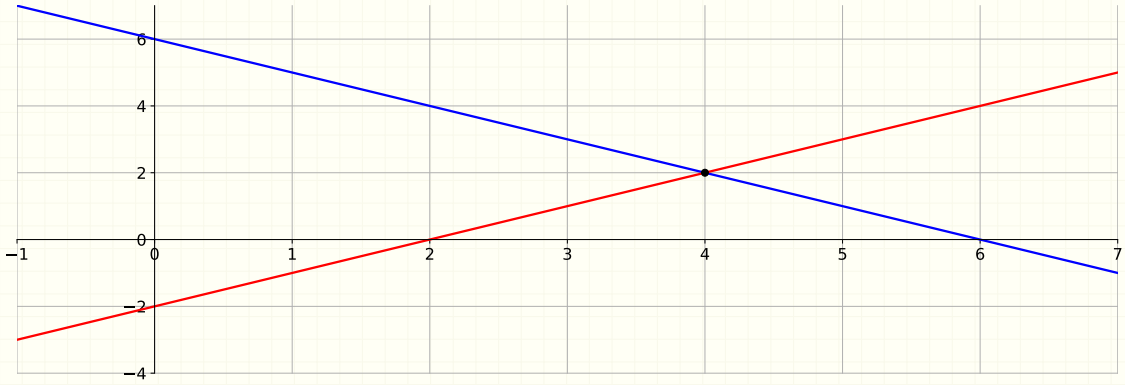

Figura 5.1: Representação geométrica do sistema (5.1).

Há somente um ponto que satisfaz as duas equações simultaneamente. Logo, o sistema (5.1) é um sistema possível e determinado.

No próximo exemplo vamos ver um caso onde o sistema é possível, porém indeterminado. Apresentando também sua representação geométrica no plano cartesiano.

EXEMPLO 5.1.2: Classifique o sistema linear abaixo através da análise da sua representação geométrica no plano cartesiano.

$$\begin{cases} x + y = 1 \\ 2x + 2y = 2 \end{cases} \tag{5.2}$$

A Figura 5.2 apresenta geometricamente esse sistema, mostrando as duas retas geradas pelas equações do sistema, e elas estão sobrepostas, ou seja, todos os pontos pertencentes a uma das retas também pertencem à outra reta (código).

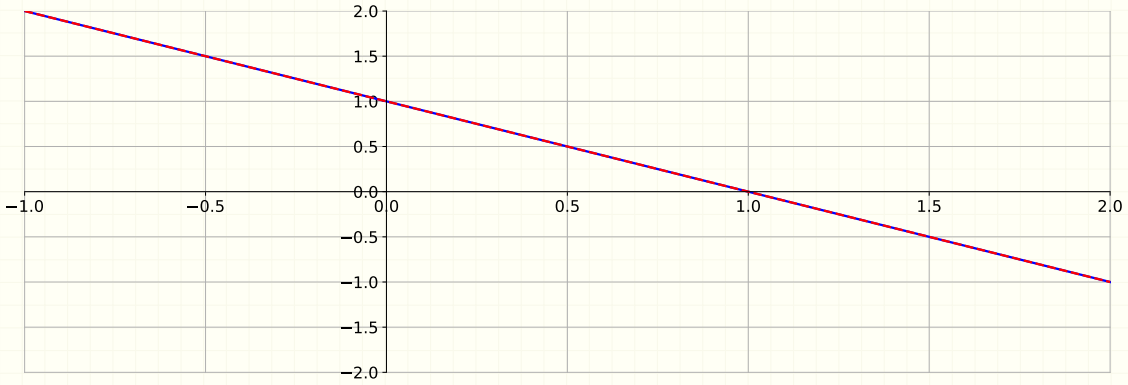

Figura 5.2: Representação geométrica do sistema (5.2).

O sistema (5.2) é possível e indeterminado, pois possui infinitas soluções.

O próximo exemplo trata de um sistema impossível, pois não há nenhum ponto em comum para ambas as retas.

Exemplo 5.1.3: Classifique o sistema linear abaixo através da análise da sua representação geométrica no plano cartesiano.

$$\begin{cases} x + y = 1 \\ x + y = 4 \end{cases} \tag{5.3}$$

A Figura 5.3 apresenta geometricamente esse sistema, mostrando as duas retas geradas pelas equações do sistema, que são paralelas entre si, ou seja, não possuem nenhum ponto em comum (código).

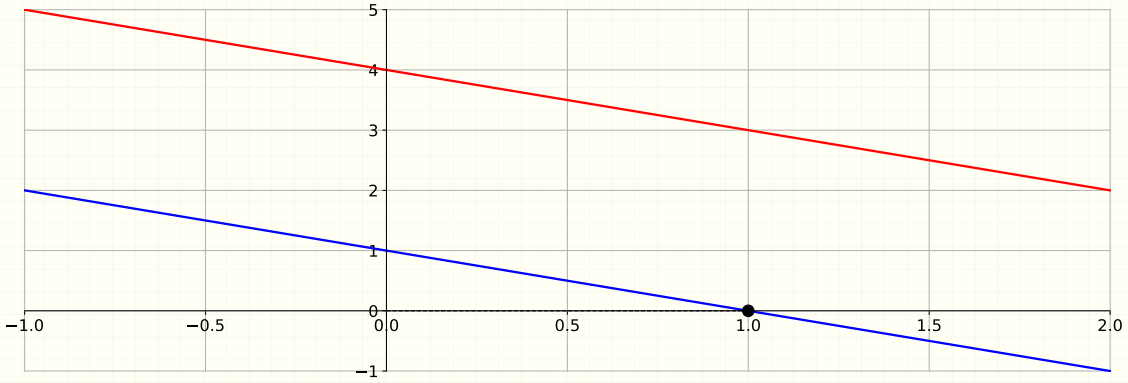

Figura 5.3: Representação geométrica do sistema (5.3)

O sistema (5.3) é impossível, pois não há nenhum ponto que satisfaça as duas equações simultaneamente.

Na matemática e na ciência, frequentemente encontramos sistemas de equações lineares que precisam ser resolvidos para solucionar problemas do mundo real. No entanto, resolver esses sistemas de forma direta e analítica nem sempre é viável, especialmente quando são grandes e complexos. É nesse contexto que os métodos numéricos se tornam essenciais. A seguir vamos explorar o Método de Jacobi, uma técnica computacional iterativa clássica que nos permite encontrar soluções aproximadas para sistemas lineares, facilitando a resolução de problemas que seriam intratáveis por métodos analíticos diretos.

Método de Jacobi

O Método de Jacobi é uma técnica iterativa usada para resolver sistemas de equações lineares, especialmente os grandes e complexos. Baseia-se na atualização sucessiva das estimativas das variáveis, utilizando as equações do sistema para aproximar as soluções. É útil quando métodos diretos são impraticáveis, permitindo obter soluções precisas através de iterações sucessivas. Para simplificar a apresentação, mostraremos os passos do método em um sistema 2×2, ou seja, sistemas com duas equações e duas variáveis, porém o método pode ser aplicado em sistemas maiores.

O passo a passo do Método de Jacobi começa com a inicialização das estimativas para x e y. Para cada iteração, isola-se x na primeira equação e utiliza-se as estimativas atuais de y para calcular um novo valor de x. Da mesma forma, isola-se y na segunda equação e usa-se as estimativas atuais de x para calcular um novo valor de y. Repete-se essas etapas até que as estimativas de x e y se estabilizem o suficiente para serem consideradas a solução. No Método de Jacobi, x e y são atualizados separadamente a cada iteração, com base nas equações do sistema e isolando cada variável em sua respectiva equação.

Considere o sistema de equações lineares a seguir:

$$\begin{cases} 2x - y = 1 \\ x + 2y = 3. \end{cases} \tag{5.4}$$

A Figura 5.4 apresenta geometricamente esse sistema, mostrando as duas retas geradas pelas equações do sistema, bem como o único ponto de interseção entre elas (código).

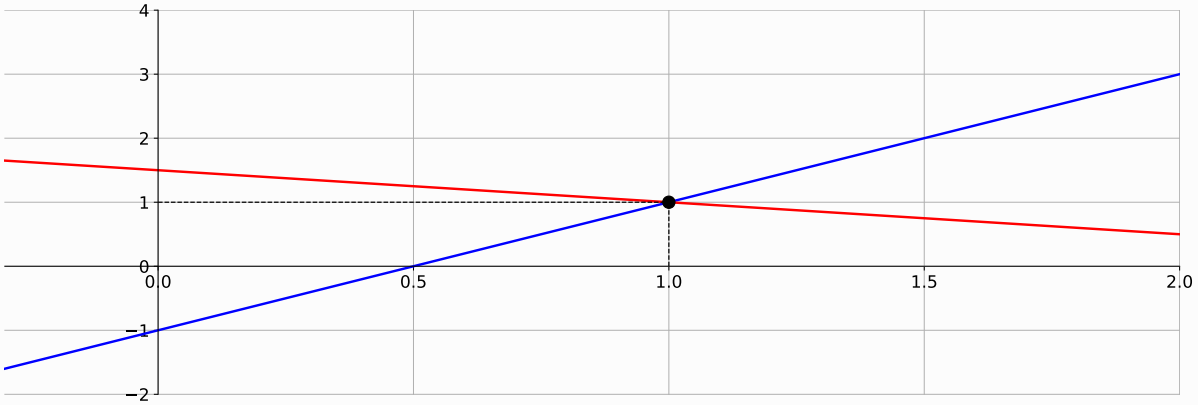

Figura 5.4: Representação geométrica do sistema (5.4).

Podemos verificar que a solução para o sistema é o par ordenado $(1, 1)$, ou seja, $x = 1$ e $y = 1$, pois substituindo esses valores nas equações do sistema

$$\begin{cases} 2 \cdot 1 - 1 = 1 \\ 1 + 2 \cdot 1 = 3, \end{cases}$$

verificamos que ambas as igualdades são verdadeiras, validando a solução.

Vamos encontrar a solução aproximada do sistema (5.4) como exemplo, aplicando o Método de Jacobi. Isso nos ajudará a entender como o método usa estimativas iniciais e as aproxima cada vez mais da solução correta do sistema.

EXEMPLO 5.1.4: Encontre uma solução aproximada para o sistema de equações lineares a seguir usando o Método de Jacobi.

$$\begin{cases} 2x - y = 1 \\ x + 2y = 3. \end{cases}$$

Vamos aplicar o Método de Jacobi para aproximar a solução desse sistema de equações lineares.

Podemos reescrever o sistema da seguinte forma:

$$\begin{cases} x = \dfrac{y + 1}{2} \\ y = \dfrac{3 - x}{2}. \end{cases}$$

Assim o processo de iteração do Método de Jacobi será:

$$x_{n+1} = \frac{y_n + 1}{2} \tag{5.5}$$

$$y_{n+1} = \frac{3 - x_n}{2}. \tag{5.6}$$

Iniciaremos o processo de iteração do método com as estimativas iniciais $x_0 = 0$ e $y_0 = 0$. Para cada iteração subsequente, utilizaremos os valores encontrados na iteração anterior para atualizar as estimativas de x e y, conforme as fórmulas das equações (5.5) e (5.6). Vamos executar 5 iterações para observar como as estimativas se aproximam da solução do sistema.

1. Primeira iteração, usando as estimativas iniciais $x_0 = 0$ e $y_0 = 0$:

$$x_1 = \frac{y_0 + 1}{2} = \frac{0 + 1}{2} = 0{,}5$$

$$y_1 = \frac{3 - x_0}{2} = \frac{3 - 0}{2} = 1{,}5.$$

2. Segunda iteração, usando $x_1 = 0{,}5$ e $y_1 = 1{,}5$:

$$x_2 = \frac{y_1 + 1}{2} = \frac{1{,}5 + 1}{2} = 1{,}25$$

$$y_2 = \frac{3 - x_1}{2} = \frac{3 - 0{,}5}{2} = 1{,}25.$$

3. Terceira iteração, usando $x_2 = 1{,}25$ e $y_2 = 1{,}25$:

$$x_3 = \frac{y_2 + 1}{2} = \frac{1{,}25 + 1}{2} = 1{,}125$$

$$y_3 = \frac{3 - x_2}{2} = \frac{3 - 1{,}25}{2} = 0{,}875.$$

4. Quarta iteração, usando $x_3 = 1{,}125$ e $y_3 = 0{,}875$:

$$x_4 = \frac{y_3 + 1}{2} = \frac{0{,}875 + 1}{2} \approx 0{,}937$$

$$y_4 = \frac{3 - x_3}{2} = \frac{3 - 1{,}125}{2} \approx 0{,}937.$$

5. Quinta iteração, usando $x_4 \approx 0{,}937$ e $y_4 \approx 0{,}937$:

$$x_5 = \frac{y_4 + 1}{2} = \frac{0{,}937 + 1}{2} \approx 0{,}968$$

$$y_5 = \frac{3 - x_4}{2} = \frac{3 - 0{,}937}{2} \approx 1{,}031.$$

Podemos observar que a cada iteração os valores se aproximam mais da solução do sistema que é $x = 1$ e $y = 1$. Repetimos o processo até atingir a aproximação desejada.

A Figura 5.5 mostra geometricamente como os resultados das iterações vão se aproximando da solução do sistema (código).

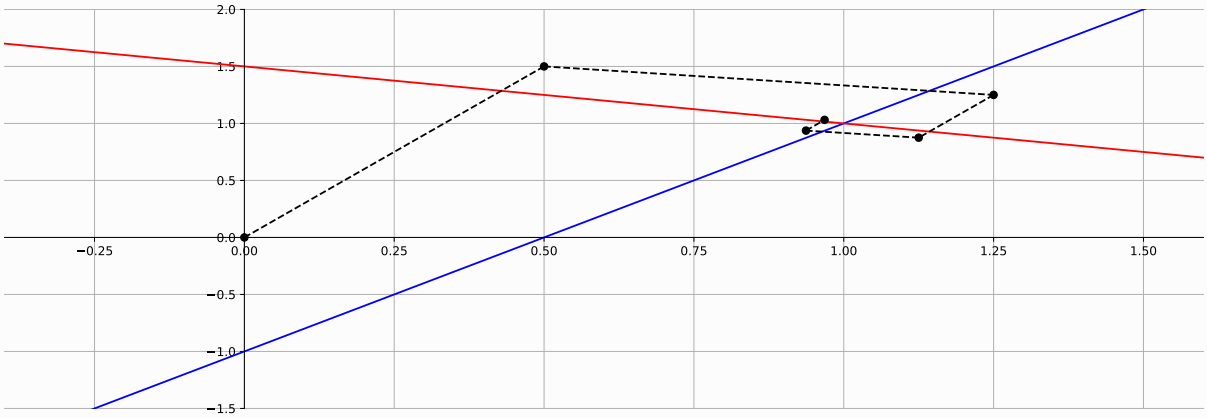

Figura 5.5: Representação geométrica do Método de Jacobi.

Para atingir uma melhor aproximação vamos escrever um programa especialmente para esse sistema usando o Método de Jacobi. Para isso o programa solicitará ao usuário a estimativa inicial para x e a estimativa inicial para y e em seguida a aproximação desejada (código).

```
1 #Programa para aproximar a solução do sistema com o Método de
    Jacobi
2
3 # Função para calcular o próximo valor de x usando o Método de
    Jacobi
4 def jacobi_iteracao(x, y):
5     x_proximo = (y + 1) / 2
6     y_proximo = (3 - x) / 2
7     return x_proximo, y_proximo
8
```

```
# Entrada das estimativas iniciais e da tolerância
x0 = float(input("Digite a estimativa inicial para x: "))
y0 = float(input("Digite a estimativa inicial para y: "))
tolerancia = float(input("Digite a tolerância desejada: "))

# Execução das iterações do Método de Jacobi
max_iteracoes = 100  # Número máximo de iterações
iteracao = 0
x = x0
y = y0
while iteracao < max_iteracoes:
    x_proximo, y_proximo = jacobi_iteracao(x, y)
    print(f"Iteração {iteracao + 1}: x = {x_proximo}, y = {y_proximo}")
    if abs(x_proximo - x) < tolerancia and abs(y_proximo - y) < tolerancia:
        break
    x = x_proximo
    y = y_proximo
    iteracao += 1

# Resultado das iterações
print(f"\nResultado após {iteracao} iterações:")
print(f"x = {x}, y = {y}")
```

O programa utiliza o Método de Jacobi para aproximar a solução de um sistema de equações. A função `jacobi_iteracao` calcula os próximos valores de `x` e `y` com base nas fórmulas do método. O usuário fornece as estimativas iniciais e a tolerância desejada através dos comandos `input`. O *loop* `while` executa o método até que a diferença entre os valores sucessivos de `x` e `y` seja menor que a tolerância ou o número máximo de iterações seja alcançado. Os novos valores são exibidos a cada iteração. Quando a precisão desejada é atingida, o programa mostra os valores finais e o número total de iterações realizadas.

Supondo que o usuário digite $x_0 = 0$ e $y_0 = 0$ para os valores iniciais e 0.001 para a tolerância, ou seja, desejando uma aproximação de 3 casas decimais de precisão, a resposta do programa será

```
Digite a estimativa inicial para x: 0
Digite a estimativa inicial para y: 0
Digite a tolerância desejada: 0.001
Iteração 1: x = 0.5, y = 1.5
```

```
Iteração 2: x = 1.25, y = 1.25
Iteração 3: x = 1.125, y = 0.875
Iteração 4: x = 0.9375, y = 0.9375
Iteração 5: x = 0.96875, y = 1.03125
Iteração 6: x = 1.015625, y = 1.015625
Iteração 7: x = 1.0078125, y = 0.9921875
Iteração 8: x = 0.99609375, y = 0.99609375
Iteração 9: x = 0.998046875, y = 1.001953125
Iteração 10: x = 1.0009765625, y = 1.0009765625
Iteração 11: x = 1.00048828125, y = 0.99951171875
Iteração 12: x = 0.999755859375, y = 0.999755859375

Resultado após 11 iterações:
x = 1.00048828125, y = 0.99951171875
```

Podemos observar que o programa encerrou as iterações para $x = 1{,}000$ e $y = 0{,}999$, ou seja, com 3 casas decimais de precisão como foi solicitado.

O Método de Jacobi, embora atualmente tenha sido substituído por técnicas mais avançadas, ainda é um ponto de partida importante para compreender a resolução de sistemas de equações lineares. Ele serviu de inspiração para o desenvolvimento de métodos numéricos iterativos mais recentes.

Na próxima seção, vamos abordar método numéricos para aproximar soluções de sistemas de equações não lineares e vamos apresentar o Método de Newton, expandindo ainda mais nosso conjunto de técnicas para lidar com diferentes tipos de problemas matemáticos envolvendo sistemas de equações.

5.2 Sistemas de Equações Não Lineares

Nesta seção, exploraremos os sistemas de equações não lineares. Diferentemente das equações lineares, cujas relações entre as variáveis são representadas por retas, as equações não lineares envolvem relações mais complexas, como quadráticas, cúbicas, exponenciais, entre outras formas. Esses sistemas são frequentemente encontrados em áreas como física, biologia, economia e engenharia. Em seguida, apresentaremos o Método de Newton, uma técnica poderosa para resolver sistemas de equações não lineares. É importante destacar que este método é uma generalização do Método de Newton, discutido no Capítulo 4, que foi originalmente desenvolvido para resolver uma única equação não linear. A generalização permite aplicar a técnica ao contexto

de sistemas de equações não lineares, oferecendo uma abordagem eficaz para lidar com a complexidade matemática desses problemas.

Método de Newton para Sistemas Não Lineares

O Método de Newton para sistemas não lineares começa com uma estimativa inicial para a solução do sistema. Com base nessa estimativa, calculamos uma nova estimativa aplicando a fórmula do método, que envolve a matriz Jacobiana, uma matriz formada pelas derivadas das funções do sistema e seu nome "matriz Jacobiana" é em homenagem ao matemático Jacobi. O método usa a fórmula iterativa para atualizar a estimativa: a nova estimativa é obtida subtraindo a inversa da matriz Jacobiana multiplicada pelo vetor das funções avaliadas na estimativa atual. Cada nova estimativa é então usada para recalcular a matriz Jacobiana e o vetor das funções, e o processo é repetido. Cada iteração geralmente aproxima a estimativa da solução correta, e o processo continua até que a diferença entre as estimativas sucessivas seja suficientemente pequena, alcançando a precisão desejada.

Todo esse conhecimento matemático geralmente faz parte do cálculo avançado, estudado no ensino superior. No entanto, vamos tentar mostrar os passos desse método com funções mais simples, para que, mesmo sem esse conhecimento avançado, seja possível entender o funcionamento do método.

Vamos considerar um sistema não linear 2×2 para ilustrar o Método de Newton. Considere o seguinte sistema:

$$\begin{cases} y = -x^2 - 2x + 7 \\ y = 2^x + 1. \end{cases} \tag{5.7}$$

Observe que este sistema é difícil de ser solucionado de forma analítica. A Figura 5.6 apresenta geometricamente esse sistema e mostra o gráfico das duas equações e os pontos de interceção entre elas. Observe que se trata de duas curvas, ou seja, equações não lineares (código).

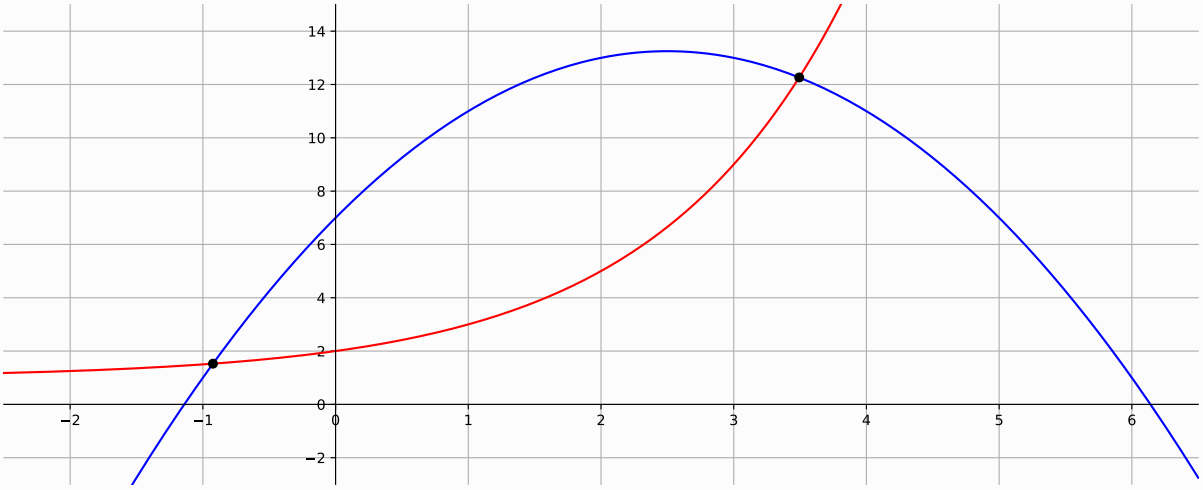

Figura 5.6: Representação geométrica do sistema (5.7).

Podemos observar que existem dois pontos de intercessão entre as curvas geradas pelas funções mostradas no gráfico, ou seja, o sistema possui duas soluções: uma onde o valor de x está entre -1 e 0 e y está entre 1 e 2, e outra onde o valor de x está entre 3 e 4 e y está entre 12 e 13. A utilização de métodos gráficos é muito cara computacionalmente. Por exemplo, para construir esse gráfico, foram necessários 400 pontos. Para funções grandes e complexas, isso pode exigir muitos recursos computacionais. Uma das vantagens dos métodos numéricos é que eles requerem uma quantidade muito menor de iterações para se aproximar da solução desejada.

Vamos, no exemplo a seguir aplicar um passo do Método de Newton para aproximar as estimativas iniciais à uma das soluções do sistema.

EXEMPLO 5.2.1: Considere o sistema não linear (5.7) dado por

$$\begin{cases} y = -x^2 + 5x + 7 \\ y = 2^x + 1. \end{cases}$$

Desenvolva um passo do Método de Newton para aproximar as estimativas iniciais $x_0 = 3$ e $y_0 = 12$, de uma das soluções do sistema.

Primeiro, definimos as funções f_1 e f_2 do sistema (5.7) como:

$$\begin{cases} f_1(x, y) = -x^2 + 5x + 7 - y \\ f_2(x, y) = 2^x + 1 - y. \end{cases}$$

Calculamos as derivadas parciais e com elas construimos a matriz Jacobiana

$$J = \begin{bmatrix} -2x + 5 & -1 \\ 2^x \ln(2) & -1 \end{bmatrix}.$$

Começamos com as estimativas iniciais $x_0 = 3$ e $y_0 = 12$ e as substituímos nas funções do sistema.

$$f_1(3, 12) = -3^2 + 5 \cdot 3 + 7 - 12 = 1$$

$$f_2(3, 12) = 2^3 + 1 - 12 = -3.$$

Substituímos $x_0 = 3$ na matriz Jacobiana.

$$J = \begin{bmatrix} -2 \cdot 3 + 5 & -1 \\ 2^3 \ln(2) & -1 \end{bmatrix} = \begin{bmatrix} -1 & -1 \\ 5.544 & -1 \end{bmatrix}.$$

A inversa da matriz Jacobiana J é

$$J^{-1} \approx \begin{bmatrix} -0.153 & 0.153 \\ -0.848 & -0.153 \end{bmatrix}.$$

A fórmula de iteração do Método de Newton para sistemas não lineares é dada por

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \end{bmatrix} - J^{-1} \begin{bmatrix} f_1(x_n, y_n) \\ f_2(x_n, y_n) \end{bmatrix}.$$

Substituímos os valores e calculando o produto, obtemos as novas estimativas.

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} 3.612 \\ 12.389 \end{bmatrix}$$

Assim, após a primeira iteração, temos as estimativas atualizadas $x_1 \approx 3.612$ e $y_1 \approx 12.389$.

Como o Método de Newton envolve vários passos e cálculos complexos, a programação é uma ferramenta eficaz para automatizar todo esse processo. O código a seguir implementa o Método de Newton para encontrar soluções aproximadas para sistemas de equações não lineares. Ele define as funções do sistema e suas derivadas parciais, utilizando uma função que aplica o Método de Newton com base em estimativas

iniciais fornecidas pelo usuário. Isso facilita a resolução prática e eficiente de sistemas complexos, proporcionando soluções numéricas de forma automatizada.

O código que realiza esse processo é apresentado abaixo (código).

```
def Metodo_Newton(eq1, eq2, partial_eq1_x, partial_eq1_y, partial_eq2_x, partial_eq2_y, x0, y0, tolerance):
    iterations = 0
    results = []

    while True:
        iterations += 1
        f1 = eq1(x0, y0)
        f2 = eq2(x0, y0)

        J = np.array([[partial_eq1_x(x0), partial_eq1_y()],
            [partial_eq2_x(x0), partial_eq2_y()]])

        inv_J = np.linalg.inv(J)
        delta = np.dot(inv_J, np.array([-f1, -f2]))
        x1 = x0 + delta[0]
        y1 = y0 + delta[1]

        results.append((x1, y1))
        print(f"x_{iterations} = {x1:.4f}, y_{iterations} = {y1:.4f}")

        if abs(x1 - x0) < tolerance and abs(y1 - y0) < tolerance:
            break

        x0, y0 = x1, y1

    return x1, y1, iterations
```

Este código define a função `Metodo_Newton`, que aplica o Método de Newton para resolver sistemas de equações não lineares. A função recebe como parâmetros as equações do sistema (`eq1` e `eq2`), suas derivadas parciais (`partial_eq1_x`, `partial_eq1_y`, `partial_eq2_x` e `partial_eq2_y`), estimativas iniciais para `x` e `y`, e a tolerância para a convergência. Em cada iteração, a função calcula os valores das equações, constrói e inverte a matriz Jacobiana para encontrar os incrementos necessários nas variáveis. Esses incrementos são aplicados até que as mudanças entre iterações sejam menores que a tolerância especificada. A função imprime as estimativas de cada iteração e retorna as estimativas finais e o número de iterações realizadas.

Agora, considerando o sistema de equações não lineares (5.7) dado por

$$\begin{cases} y = -x^2 + 5x + 7 \\ y = 2^x + 1, \end{cases}$$

vamos escrever um código para definir as funções que compõem o sistema de equações e suas funções derivadas e especificar as estimativas iniciais e a tolerância desejada. Em seguida, utilizaremos a função Método de Newton, que foi definida anteriormente, para encontrar as soluções aproximadas desse sistema.

O código que realiza esse processo é apresentado abaixo (código).

```
1  import math
2
3  # Funções do sistema de equações
4  def eq1(x, y):
5      return -x**2 + 5*x + 7 - y
6
7  def eq2(x, y):
8      return 2**x + 1 - y
9
10 # Derivadas parciais das funções
11 def partial_eq1_x(x):
12     return -2*x + 5
13
14 def partial_eq1_y():
15     return -1
16
17 def partial_eq2_x(x):
18     return 2**x * math.log(2)
19
20 def partial_eq2_y():
21     return -1
22
23 # Coleta de dados do usuário e chamada da função
24 if __name__ == "__main__":
25     x0 = float(input("Digite a estimativa inicial para x: "))
26     y0 = float(input("Digite a estimativa inicial para y: "))
27     tolerance = float(input("Digite a tolerância para as
       aproximações: "))
28
29     x_final, y_final, num_iterations = Metodo_Newton(
30         eq1, eq2,
```

```
        partial_eq1_x, partial_eq1_y,
        partial_eq2_x, partial_eq2_y,
        x0, y0, tolerance
    )

    print(f"\nApós {num_iterations} iterações temos:")
    print(f"x_{num_iterations} = {x_final:.4f}")
    print(f"y_{num_iterations} = {y_final:.4f}")
```

Este código coleta as entradas do usuário e chama a função `Metodo_Newton`. Ele solicita ao usuário as estimativas iniciais para `x` e `y`, e a tolerância desejada. As funções do sistema de equações e suas derivadas parciais são definidas conforme o sistema a ser resolvido. Após coletar os dados, o código chama a função `Metodo_Newton` com os parâmetros fornecidos e exibe o resultado final, incluindo a solução para `x` e `y` e o número de iterações necessárias para alcançar a precisão desejada.

Vamos iniciar tentando aproximar a solução onde o valor de x se encontra no intervalo entre 3 e 4. Para isso vamos iniciar com as estimativas $x_0 = 3$ e $y_0 = 12$ e com uma aproximação de 3 casas decimais. Após a execução, o programa gera os resultados a seguir.

```
Digite a estimativa inicial para x: 3
Digite a estimativa inicial para y: 12
Digite a tolerância para as aproximações: 0.001
x_1 = 3.6111, y_1 = 12.3889
x_2 = 3.4985, y_2 = 12.2657
x_3 = 3.4935, y_3 = 12.2629
x_4 = 3.4935, y_4 = 12.2629

Após 4 iterações temos:
x_4 = 3.4935
y_4 = 12.2629
```

Após 4 iterações o programa encontrou os resultados aproximados $x = 3{,}493$ e $y = 12{,}263$. Geometricamente a Figura 5.7 mostra como os valores obtidos pelas iterações vão se aproximando da solução do sistema (código).

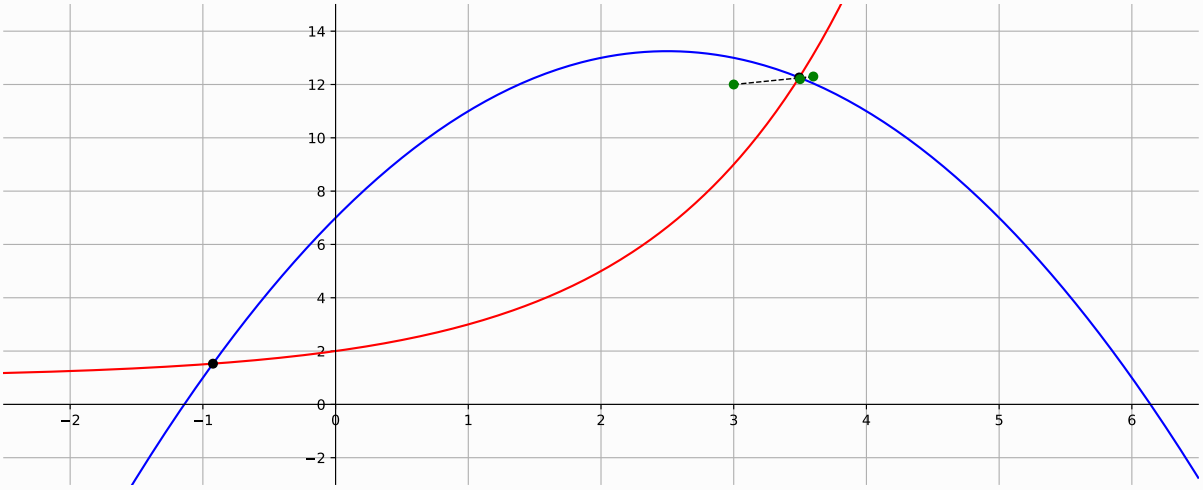

Figura 5.7: Representação geométrica do sistema (5.7) e dos pontos gerados pelo Método de Newton.

Agora vamos iniciar tentando aproximar a solução onde o valor de x se encontrar no intervalo entre -1 e 0. Para isso vamos iniciar com as estimativas $x_0 = 0$ e $y_0 = 0$ e com uma aproximação de 3 casas decimais. Com esses dados iniciais, o programa gera os resultados a seguir.

```
Digite a estimativa inicial para x: 0
Digite a estimativa inicial para y: 0
Digite a tolerância para as aproximações: 0.001
x_1 = -1.1609, y_1 = 1.1953
x_2 = -0.9328, y_2 = 1.5179
x_3 = -0.9239, y_3 = 1.5271
x_4 = -0.9239, y_4 = 1.5271

Após 4 iterações temos:
x_4 = -0.9239
y_4 = 1.5271
```

Após 4 iterações o programa encontrou os resultados aproximados $x = -0{,}924$ e $y = 1{,}527$. Geometricamente a Figura 5.8 mostra como os valores obtidos pelas iterações vão se aproximando da solução do sistema (código).

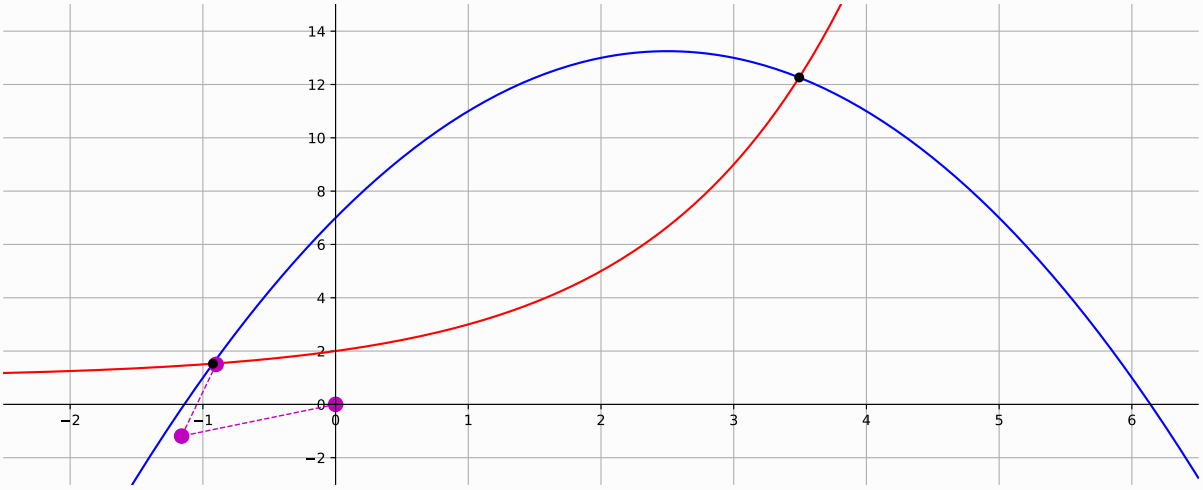

Figura 5.8: Representação geométrica do sistema (5.7) e dos pontos gerados pelo Método de Newton.

Agora, utilizando os programas que desenvolvemos, vamos aplicar o Método de Newton para resolver um exemplo prático de sistema não linear. Este exemplo ilustrará como as soluções aproximadas podem ser obtidas e aplicadas a problemas reais, demonstrando a eficácia do método em contextos práticos.

EXEMPLO 5.2.1: Dois países estão em guerra, um deles lança um míssil que sai da posição $x = 9$ se se movimentando segundo a função $f(x) = -x^2 + 12x - 27$ e tem por objetivo atingir o ponto $x = 3$ de território inimigo. O outro país por sua vês lança um míssil interceptador saindo da posição $x = 0$ se movimentando segundo a função $f(x) = -x^2 + 4x$. Encontre o ponto onde os mísseis se encontraram (código).

O problema se trata de encontrar a solução de um sistemas de equações não lineares escrito como

$$\begin{cases} y = -x^2 + 4x \\ y = -x^2 + 12x - 27 \end{cases}$$

O código que realiza esse processo é apresentado abaixo.

```
1 import numpy as np
2 import math
3
4 # Funções do sistema de equações
5 def eq1(x, y):
```

```
    return -x**2 + 4*x - y

def eq2(x, y):
    return -x**2 + 12*x - 27 - y

# Derivadas parciais das funções
def partial_eq1_x(x):
    return -2*x + 4

def partial_eq1_y():
    return -1

def partial_eq2_x(x):
    return -2*x + 12

def partial_eq2_y():
    return -1

# Função Metodo_Newton foi definida em outro lugar

# Coleta de dados do usuário e chamada da função
if __name__ == "__main__":
    x0 = float(input("Digite a estimativa inicial para x: "))
    y0 = float(input("Digite a estimativa inicial para y: "))
    tolerance = float(input("Digite a tolerância para as
   aproximações: "))

    # Chamada da função Metodo_Newton
    x_final, y_final, num_iterations = Metodo_Newton(
        eq1, eq2,
        partial_eq1_x, partial_eq1_y,
        partial_eq2_x, partial_eq2_y,
        x0, y0, tolerance
    )

    print(f"\nApós {num_iterations} iterações temos:")
    print(f"x_{num_iterations} = {x_final:.6f}")
    print(f"y_{num_iterations} = {y_final:.6f}")
```

Esse código é similar ao código do exemplo anterior, mudando apenas as funções e suas derivadas. Nesse programa será solicitado ao usuário a estimativa iniciais para `x` e `y` e em seguida a aproximação desejada. O programa chamará a função

`Metodo_Newton` que executará os passos de iteração do Método de Newton até que se obtenha a aproximação desejada.

Para a execução do programa, vamos iniciar com as estimativas $x_0 = 3$ e $y_0 = 0$ que é o ponto de impacto do míssil agressor, e com uma aproximação de 6 casas decimais. Após a execução, o programa apresentou os resultados a seguir.

```
Digite a estimativa inicial para x: 3
Digite a estimativa inicial para y: 0
Digite a tolerância para as aproximações: 0.000001
x_1 = 3.3750, y_1 = 2.2500
x_2 = 3.3750, y_2 = 2.1094
x_3 = 3.3750, y_3 = 2.1094

Após 3 iterações temos:
x_3 = 3.375000
y_3 = 2.109375
```

O programa executou 3 iterações obtendo os resultados $x = 3{,}375$ e $y = 2{,}109375$.

A Figura 5.9 mostra geometricamente a movimentação dos dois mísseis e como as iterações do Método de Newton se aproximam cada vez mais do ponto de encontro entre eles (código).

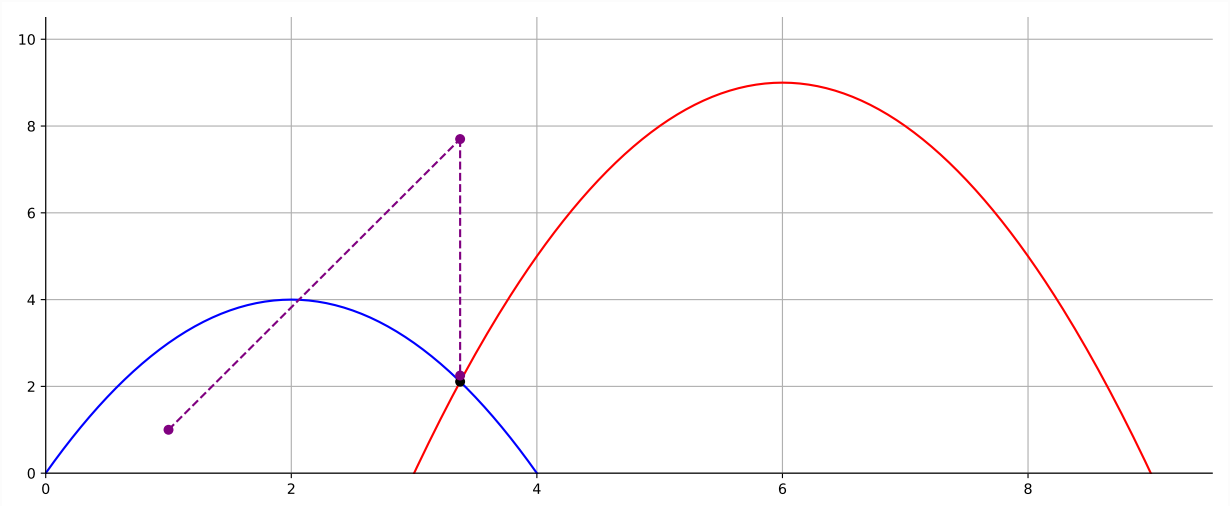

Figura 5.9: Representação geométrica do movimento dos mísseis e dos pontos gerados pelo Método de Newton.

Ao concluir a seção sobre o Método de Newton para sistemas não lineares, é importante ressaltar sua eficácia em resolver equações complexas. Adaptando esse método, podemos abordar uma ampla gama de problemas matemáticos e científicos com relações não lineares. Sua aplicação é importante em várias áreas da ciência e engenharia, superando as limitações dos métodos tradicionais e oferecendo soluções aproximadas de maneira eficiente e precisa.

Encerrando o capítulo sobre métodos iterativos para sistemas lineares e não lineares, podemos destacar a importância dessas técnicas na resolução de problemas complexos e de grande escala. Os métodos iterativos, como o Método de Jacobi e o Método de Newton, proporcionam ferramentas poderosas para encontrar soluções aproximadas de sistemas de equações, sejam eles lineares ou não lineares. Sua eficiência e versatilidade tornam essas técnicas indispensáveis em diversas áreas da matemática aplicada, ciências naturais, engenharias e outras disciplinas que lidam com modelagem e análise de sistemas complexos.

6

Considerações Finais

Ao longo deste trabalho, exploramos desde os conceitos básicos de Python até a aplicação prática de métodos numéricos na resolução de problemas matemáticos, como zeros de funções e sistemas de equações. Python facilita a aprendizagem da programação e permite a implementação direta de algoritmos matemáticos, como o cálculo de raízes quadradas e a solução de sistemas de equações.

As funções são fundamentais tanto na matemática quanto na programação em Python, onde, em Python, elas são blocos de código reutilizáveis que executam tarefas específicas, permitindo uma organização e estruturação mais eficientes do código. Compreender o conceito teórico de funções é essencial, pois isso facilita a implementação prática em Python, tornando a resolução de problemas matemáticos mais clara e eficaz. Essa combinação de teoria e prática é crucial para qualquer programador que busca aplicar a matemática em suas soluções.

Métodos numéricos, como o Método de Newton, são essenciais para encontrar soluções de problemas matemáticos de forma computacional. O Método de Newton, por exemplo, é eficiente na busca por zeros de funções, utilizando uma abordagem iterativa com rápida convergência, desde que a condição inicial seja adequada. A Bisseção, apesar de mais lenta, é um método simples e exige menos critérios para convergência, e o Método da Secante, uma variação do Método de Newton, é útil quando a derivada da função não está disponível.

Na resolução de sistemas lineares, o Método de Jacobi é uma técnica iterativa de fácil entendimento e aplicação, enquanto para sistemas não lineares, o Método de Newton se mostra uma ferramenta poderosa, utilizando a matriz Jacobiana para alcançar a convergência.

Este material oferece uma base para que você aplique esses conhecimentos em suas próprias pesquisas e projetos futuros. Esperamos que este conteúdo tenha proporcionado tanto a teoria quanto a prática necessárias para seu início na programação e na matemática computacional. Boa sorte em sua jornada!

Métodos Numéricos e Python no Ensino Médio

Edson Bertosa Lopes Santos
Luis Alberto D'Afonseca
Carlos Magno Martins Cosme

Centro Federal dealign Educação Tecnológica de Minas Gerais – CEFET-MG
19 de dezembro de 2024

Esta apostila é produto do mestrado de Edson Bertosa Lopes Santos defendida em 2024 no Profmat do Cefet-MG.

A versão mais recente desta apostila pode ser baixada clicando ou escaneando o código QR.

Arte da capa: Fotografia de Designdrunkard baixada de Pexels