ICMC-USP

Notas de Aula
SME0332

Fundamentos de Programação de Computadores

com Aplicações em python para Física e Bioinformática

Roberto F. Ausas

November 27, 2025

Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo

Prefacio

Este documento está organizado por capítulos, cada um dos quais apresenta conceitos e ferramentas essenciais para desenvolver programas de computador na linguagem `python`. Os problemas e exemplos que serão desenvolvidos terão uma ênfase nas aplicações em física e bioinformática. Em cada capítulo serão apresentados problemas e atividades que o aluno precisará desenvolver, com as quais, este será avaliado ao longo do curso.

Que tópicos este curso apresenta

Ao longo do curso iremos estudar os seguintes temas listados na sequência:

Capítulo |01| Um primeiro contato à Programação em `python`;
Capítulo |02| Cálculos estocásticos;
Capítulo |03| Processos Iterativos e Relaxações;
Capítulo |04| Outros tópicos de programação em `python`;

RFA

Contents

UM PRIMEIRO CONTATO À PROGRAMAÇÃO EM `python` 1

1.1 Preludio

A principal ideia por trás de aprender uma linguagem de programação é a de automatizar certos cálculos que surgem em física ou engenharia, que não poderiam ser resolvidos manualmente, sem o auxilio de um computador.

Nas primeiras aulas precisamos introduzir alguns conceitos básicos de programação (que se aplicam a qualquer linguagem) e posteriormente precisamos introduzir a sintaxe específica da linguagem que vamos utilizar ao longo do curso, que é a linguagem `python`.

De maneira muito geral, as linguagens de programação, podem ser divididas em duas categorias:

- **Linguagens compiladas:** Dentre as primeiras temos linguagens tais como `C`, `C++` e `Fortran`. A escrita de código neste tipo de linguagens de programação requer um domínio maior da sintaxe e das funcionalidades da linguagen. Como contrapartida, este tipo de linguagens produzem códigos que são muito rápidos pois tem sido optimizadas ao longo dos anos, e por tanto são usadas amplamente na Engenharia. De fato, a maioria dos códigos de cálculo usados na indústria (*Open Source* ou comerciais) estão feitos com elas. Ao **compilar** o código e gerar um arquivo binário optimizado, é possivel tirar o maior proveito do poder de processamento de um computador.

- **Linguagens interpretadas:** Por outra parte, as linguagens interpretadas, como `python` e `Matlab`, são relativamente simples e intuitivas, pela sua flexibilidade na sintaxe, tornando o processo de desenvolvimento mais rápido e ágil. De maneira grosseira, o código vai sendo executado a medida que se **interpreta**. Porém, se não são tomados alguns recaudos no desenho do código, a performance computacional delas pode estar bastante aquém do necessário para resolver problemas de grande porte. Um exemplo prototípico disto, é quando utilizamos uma estrutura de repetição (como um `for`) no qual realizamos um grande número de operações em cada passo. Ao longo de curso iremos chamando a atenção sobre isto, tentando introduzir boas práticas de programação.

Como comparação, vejamos o exemplo de um código simples para criar um array com números de ponto flutuante de dupla precisão, popular ele com os números $0, \ldots, N$ e imprimir o resultado na tela, usando a linguagem compilada `C` (acima) e a linguagem interpretada `python` (embaixo).

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  int i, N = 10;
  double *array;
  array=(double *) malloc(N*sizeof(double));
  for(i=0; i < N; i++) {
       array[i] = (double) i;
       printf("%lf\n", array[i]);
     }
  free(array);
  return 0;
}
```

```python
import numpy as np
N = 10
array = np.arange(N)
print(array)
```

Olhando para o exemplo, já vemos que uma linguagem interpretada como `python` se torna mais prática para um primeiro curso de programação, pois o objetivo é desenvolver a capacidade de programar algoritmos para resolver problemas práticos no computador, sem gastar muito tempo no desenvolvimento de código.

Neste curso iremos adotar a linguagem `python`, a qual tem-se tornado bastante popular nos últimos anos. Para facilitar a implementação dos programas para resolver problemas de interesse, contaremos com algumas bibliotecas específicas que facilitarão o trabalho:

- `numpy`: Para manipulação eficiente de matrizes e vetores, operações de algebra linear computacional e vários métodos numéricos.
- `scipy`: Para métodos numéricos mais avançados ou específicos, não cobertos pela anterior.
- `matplotlib`: Para plotagens e geração de gráficos em 2D e 3D.

1.2 Noções iniciais de python

Ao longo do curso iremos incorporando diversas ferramentas e funções disponíveis em várias bibliotecas, mas antes vamos a introduzir alguns conceitos essenciais de programação específicos de `python`. Se sugere ir digitando em algum editor de código ou algum ambiente de programação.

1.2.1 Tipos de variáveis

Alguns dos tipos de variáveis mais usados são apresentados na sequência:

Números

São os objetos mais simples. Podem ser inteiros, de ponto flutuante, complexos e booleanos, por exemplo:

```
1, -2, 3.1415, 6.02e23, 1 + 1j, True, False
```

Strings

São basicamente, listas de caracteres e se escrevem entre aspas simples ou duplas:

```
"a", "USP", "21", "AbCdE", "---"
```

Listas

Servem para agrupar vários objetos.

```
mylist = [1, 3.14, "USP", True, -10000, 1+1j, myfunc]
```

A lista é indexada simplesmente pela posição do objeto, começando desde 0, p.e.,

`mylist[1]` é `3.14`
`mylist[6]` é myfunc

Diccionarios

É uma forma mais prática de definir listas com identificadores:

```
mydic = {"valor": 3.14, "minhauni": "USP", "lista": ["a", 2,
1+1j]}
```

Então,

`mydic{"minhauni"}` é `"USP"`
`mydic{"lista"}[2]` é `1 + 1j`

1.2.2 Estruturas condicionais

Uma das estruturas de programação mais usadas é a estrutura condicional. A sintaxe é simples:

```
if (Expressão lógica):
    .
    .
else:
    .
    .
```

ou em situações com mais de duas opções para decidir:

```
if (Expressão lógica 1):
```

```
    .
    .
elif (Expressão lógica 2):
    .
    .
else:
    .
    .
```

Notar os : no final das sentencias e a indentação dentro de cada bloco.

Advertência

A indentação é fundamental em python. Se esta não for respeitada o interpretador não saberá quais instruções ficam dentro da estrutura e por tanto o programa poderá ter comportamentos não esperados ou em alguns casos parar a execução .

1.2.3 Estruturas de repetição

Existem duas estruturas de repetição que são muito usadas. A primeira estrutura é o famoso for (o "para" em portugués):

```
for i in range(N):
    .
    .
```

A segunda estrutura de repetição que iremos usar as vezes é o while (o "enquanto" em portugués):

```
while (Expressão lógica):
    .
    .
```

Com elas podemos fazer qualquer tipo de cálculo em que precisamos iterar sobre os elementos de algum objeto ou repetir uma operação várias vezes.

1.2.4 Funções

As declaração de funções é fundamental para poder organizar um código, encapsulando uma serie de operações as quais pode ser necessário realizar muitas vezes. A sintaxe para declarar uma função é:

```
def minhafunc(arg1, arg2, ...):

    .

    .

    return var1, var2, ...
```

Novamente, notar os : no final da definição e a indentação dentro do bloco da função.

1.2.5 A biblioteca numpy

Esta é uma das bibliotecas que mais serão usadas ao longo do curso. Basicamente permite definir vetores, matrizes e tensores em geral, populados com números ou dados de um tipo homegêneo (i.e., todos números inteiros, ou todos números de ponto flutuante, etc.). Esta biblioteca é fundamental para poder realizar cálculos científicos com uma eficiencia razoável, possivelmente similar à da uma linguagem compilada. Alguns exemplos de uso na sequência:

```
import numpy as np


vec_ints = np.array([1,2,3,4], dtype=np.int32)
vec_doubles = np.array([1,2,3,4], dtype=np.float64)
x = np.linspace(start, end)
y = np.sin(x)
vetor_nulo = np.zeros(10, dtype=np.int32)
matriz_nula = np.zeros(shape=(5,3), dtype=np.float64)
```

A ideia era mostrar alguns exemplos simples para o aluno se familiarizar um pouco com a sintaxe. Na sequência colocaremos os conceitos em prática desenvolvindo códigos para resolver vários problemas.

1.3 Lista 1: Rudimentos básicos de programação

A melhor forma para aprender a programar é resolver problemas concretos. Na sequência temos a primeira lista de exercícios para desenvolver códigos em python.

Exercícios preliminares para praticar e não serão avaliados

1. Fazer um código que define dois vetores vec1 e vec2 de tamanho 5 com números de ponto flutuante inventados. Depois cria um outro vetor de números inteiros do mesmo tamanho, tal que na i-éssima posição ele toma o valor 1 se a componente correspondente de vec1 é maior que a do vec2 e 0 em caso contrário.

2. Fazer um código que define um ponto de $\mathbf{R}^2$ e imprime `True` ou `False` dependendo se o ponto está emcima ou embaixo da reta $y = x$.
3. Calcular o número π usando a serie:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

   Comparar o resultado dependendo do número de termos usados na série com o verdadeiro valor de π que pode ser obtido usando `np.pi`.
4. Declarar uma lista que possui nomes de pessoas inventados, outra lista que possui os seus sobrenomes e outra lista que possui a suas idades. Depois gerar listas individuais associadas a cada pessoa, na qual se encapsulem todos os dados dessa pessoa. Para isto, pode ser util usar uma propriedade das listas que é a possibilidade de acrescentar elementos usando a função `append`, por exemplo:

```
minha_lista = []            # Lista vazia
minha_lista.append('kkk')   # Acrescento o string kkk
minha_lista.append('jjj')   # Acrescento o string jjj
```

   que cria a lista `['kkk', 'jjj']`.
5. Fazer um programa que cria 10 pontos inventados em $\mathbf{R}^2$ e imprime quantos desses pontos estão fora do círculo de raio 1 centrado na origem e quantos dentro.
6. Repetir o exercício anterior para o caso de pontos em $\mathbf{R}^3$ e uma esfera.
7. Repetir o exercício anterior para o caso em que a esfera está centrada em $(0.25, 0.5, 0.75)$.
8. Fazer um script que define um vetor randômico de dimensão N e calcula a média dos valores das suas componentes, i.e.,

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

   Considerar $N = 10, 100, 1000, 10000, 100000, 1000000$. Pode usar a função de `numpy` `np.random.rand(N)` que irá criar um `array` unidimensional (i.e., um vetor), com N números randômicos com valores entre 0 e 1. Estes número possuim o que se chama uma distribuição uniforme, pois qualquer um tem a mesma chance de sair.
9. Fazer um script que define uma matriz randômica de dimensão $N \times N$ e calcula a média dos valores dos seus elementos, i.e.,

$$\bar{A} = \frac{1}{N \times N} \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ij}$$

   tomando $N = 10, 100, 500, 1000$. Pode usar a função de `numpy` `np.random.rand(N,N)` que irá criar um `array` bidimensional (i.e., uma matriz) com N linhas e N colunas com de números randômicos com valores entre 0 e 1 distribuidos uniformemente.
10. Fazer uma função que calcula o produto escalar de dois vetores

randômicos **a** e **b** de $\mathbb{R}^3$, calcula a sua magnitude e determina o ângulo θ que eles formam, usando a fórmula:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \, \cos\theta$$

11. Fazer uma função que calcula o produto vetorial de dois vetores randômicos **a** e **b** de $\mathbb{R}^3$

$$\mathbf{a} \times \mathbf{b}$$

12. Nos 5 problemas anteriores, encapsular os códigos em funções que possam ser chamadas, as quais recebam os argumentos de entrada apropriados e retornem o que é pedido em cada caso.

Gráficos simples

13. Considerar 10 dados que são jogados, podendo sair números do 1 até o 6. A soma dos valores será

$$S = \sum_{i=1}^{10} d_i$$

em que d_i é o que saiu em cada um dos dados. Jogar os 10 dados 100000 vezes e construir um histograma que mostre o comportamento de S. Um histograma seria um gráfico de frequencia de um certo evento, ou seja, quantas vezes a soma deu 10, quantas vezes a soma deu 11, quantas vezes a soma deu 12, ..., quantas vezes a soma deu 60. Para isto precissará usar a função

```
counts, bins = np.histogram(s)
plt.stairs(counts, bins)
```

em que `s` será um vetor que guardou o resultados das 100000 realizações. Pode usar a função `np.random.randint(1,7,10)` que irá gerar 10 número inteiros entre 1 e 6, com distribuição uniforme (ou seja, os dados não estão cargados)

14. Fazer um gráfico da função $f(x) = x^m$ para diferentes valores de m considerando o intervalo $x \in [1, 4]$. Usar escala linear e escala loglog.
15. Fazer um gráfico da função $f(x) = \sin(m\,x)$ para diferentes valores de m considerando o intervalo $x \in [0, 2\,\pi]$.
16. Fazer um gráfico que mostra pontos distribuídos randomicamente na região do plano $[0, 2] \times [-2, 1]$.

Aviso importante

O material de estudo para desenvolver a primeira lista será a presente apostila e os `jupyter notebooks` desenvolvidos pelo professor em sala de aula, os quais foram disponibilizados no repositório da disciplina. Outras fontes de informação a ser consideradas são os sites das bibliotecas que iremos utilizar, tais como https://numpy.org e https://matplotlib.org/, assim como consultas dirigidas ao professor ou aos monitores em forma presencial ou por e-mail (`rfausas@icmc.usp.br`).

A lista na sequência deve estar pronta para o dia 09/09. A partir desta data, o professor chamará aleatóriamente a cada aluno para explicar os exercícios.

Exercícios que serão avaliados

1. Fazer uma função que:

   - Pega dois vetores randômicos **a** e **b** de dimensão n, e dois escalares randômicos α e β e calcula um vetor **c** tal que

     $$\mathbf{c} = \alpha\mathbf{a} + \beta\mathbf{b}$$

   - Pega uma matriz randômica **A** de $n \times n$ e calcula a sua m-essima potência

     $$\mathbf{A}^m = \underbrace{\mathbf{A} \ldots \mathbf{A}}_{m \text{ vezes}}$$

     (tomar valores de $m = 2, 3, 4$).

   Em todos os casos medir o tempo necessário para realizar as operações para diferentes dimensões n. Plotar o tempo de cálculo como função da dimensão n usando a escala linear padrão e a escala `loglog`. No segundo ponto, colocar no mesmo gráfico os resultados para os diferentes valores de m. Tirar conclusões.

   Nota: Para que os resultados sejam interessantes, no primeiro ponto, tomar valores de $n = 10^4,\ 10^5, 10^6, 10^7$. Já, no segundo ponto, tomar valores de $n = 500, 1000, 1500$.

2. Fazer uma função que recebe uma matriz randômica `A` de dimensão $m \times r$ e outra matriz randômica `B` de dimensão $r \times n$, verifica as suas dimensões e realiza a multiplicação delas no sentido usual de algebra linear, para retornar uma matriz `C` de dimensão $m \times n$. A multiplicação deve ser feita usando todos os `for` que sejam necessários, lembrando que a fórmula para calcular o coeficiente c_{ij} é dado por

   $$c_{ij} = \sum_{k=1}^{r} a_{ik}\, b_{kj}, \quad i = 1, \ldots, m,\ j = 1, \ldots, n$$

   Considerando matrizes quadradas (i.e., $m = r = n$), medir o tempo de cálculo como função da dimensão e comparar com o tempo necessário fazendo `A @ B`. Para que os resultados sejam interessantes tomar dimensões de matriz $50, 100, 150, 200, 250$, como anteriormente. Novamente, plotar o tempo de cálculo como função da dimensão na escala `loglog`.

3. Fazer um gráfico que mostra pontos distribuídos randomicamente dentro da região definida por $x = 0$, $y = 0$ e $y = 1 - x$. Mostrar diferentes casos, considerando diferentes quantidades de pontos.

4. Considerar o problema do cálculo do número π usando a serie da lista anterior:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

Comparar o resultado dependendo do número de termos usados na série com o verdadeiro valor de π que pode ser obtido usando `np.pi`. Fazer um gráfico do erro no cálculo (i.e., $|\pi - \pi_N|$) como função de N. Usar escala `loglog` e escala linear.

5. **Mapeo logístico:** Considerar uma sequência de numéros gerada da seguinte forma:

$$x_n = a\, x_{n-1}\,(1 - x_{n-1}), \quad n = 1, 2, \dots, N$$

Considerar $N = 5000$, $x_0 = 0.1$ e diferentes valores de a entre 0 e 4 (p.e., $a = 1,\ 2,\ 3.2,\ 3.5,$ e 4). Calcular a média e a variança da sequência:

$$\bar{x} = \frac{1}{N}\sum_{i=0}^{N} x_i, \quad \sigma = \frac{1}{N-1}\sum_{i=0}^{N}(x_i - \bar{x})^2$$

Programar-lo na mão e usando funções de `numpy` já prontas (`np.mean` e `np.var`). Comparar os resultados.

6. No problema 5, plotar a sequéncia de valores obtida em cada caso considerando os diferentes valores de a pedidos. Fazer os gráficos usando legendas, labels, e outros atributos que achar interessante, para melhor ilustrar os resultados.

7. Continuando com a sequência do problema 3, fazer um código que gera o diagrama de bifurcações, que é um gráfico que mostra os valores que assume a sequência, como função dos valores de a. O resultado deveria ser algo do tipo mostrado na figura ao lado, que no eixo horizontal tem os valores de a usados para gerar a sequência e no eixo vertical todos os possíveis valores que toma a sequência para o correspondente valor de a. Se sugere usar pontinhos bem pequenos para gerar o gráfico. Explicar os resultados se auxiliando com os gráficos do exercício anterior.

8. Fazer uma função que plota uma região poligonal fechada. Esta função deve receber uma lista de pontos no plano x–y, desenhar os pontos com cor vermelho e linhas unindo os pontos com cor azul. A lista de coordenadas dos pontos que o usuario deseja plotar deve ser escrita num arquivo de texto, e.g.,

```
0.0 0.0
0.0 0.1
0.1 0.0
...
...
```

e para carregar o arquivo deverá ser usado o comando
`meuspontos = np.loadtxt('meuarquivo.txt')`

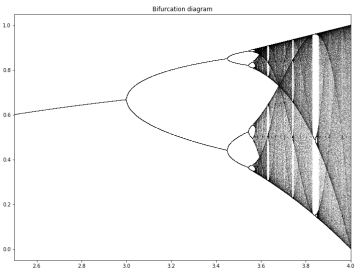


Figure 1.1: Diagrama de bifurcações.

9. Considerando que uma string pode ser vista como uma lista de caracteres, e.g.,

   ```
   meustring = 'Hola'
   print(meustring[0])
   ```

   imprime `'H'` e assim por diante. Fazer uma função que toma dois strings, compara eles, caracter por caracter e retorna, o tamanho de cada string e quantos caracteres são iguais entre eles.

10. Considerar o lançamento de uma bala de canhão, problema clássico estudado em Física 1. Fazer um programa de python que recebe o ângulo e velocidade de lançamento da bala e plotar a trajetória da bala. O código deveria funcionar para poder plotar no mesmo gráfico a trajetória correspondente a vários valores dos parâmetros de entrada.

11. Consider uma partícula que começa na posição $(0, 0)$ e começa executar uma serie de pasos. Em cada passo ela pode pula uma distancia Δ para esquerda, para direita, para cima ou para baixo. Todas as possibilidades são igualmente prováveis (i.e., $p = \frac{1}{4}$). Desenhar várias trajetórias de partículas que realizam 1000 passos. Para escolher a direção usar a função `np.random.randint(4)` que irá gerar numéros inteiros igualmente prováveis entre 0 e 3 inclusive, o que servirá para escolher a direção do pulo em cada passo.

12. **Redes e grafos:**

    Neste exercicio não pode usar classes nem bibliotecas para gerenciamento de grafos, apenas o que foi ensinado na aula e a biblioteca `numpy`.

    Considerar uma rede de distribuição com a mostrada na figura ao lado. Esta pode ser um exemplo de uma rede elétrica ou hidráulica e é basicamente o que se chama um *grafo*. Notar que em geral ela estará caracterizada por um certo número de *nós* (ou uniões), um certo número de *arestas* e alguma informação sobre a conectividade entre pontos.

    - Fazer uma estrutura de dados que sirva para descrever essa rede. Idealmente, a estrutura deveria incluir algum tipo de matriz ou *array* que indica como os nós e as arestas estão relacionados. Adicionalmente, a estrutura deve conter um array para descrever as coordenadas (x, y) de cada nó. Construir um exemplo inventando as coordenadas e plotar a rede.
    - Fazer uma função que insere uma nova aresta na rede para conectar dois pontos já existentes.
    - Fazer uma outra função que permita apagar (ou *deletar*) uma aresta da rede. Notar que se a aresta possui um nó que náo pertence a nenhuma outra aresta, esse nó também precisa ser deletado.

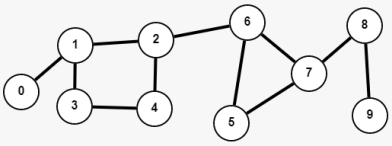


Figure 1.2: Exemplo de uma rede.

- **(Opcional)** Fazer uma função que deleta um nó e todas as arestas que emanam dele.

13. **(opcional) O jogo da vida de Conway:** O Jogo da vida é uma grade ortogonal bidimensional de células quadradas, cada uma das quais está em um dos dois estados possíveis: viva ou morta. Cada célula interage com seus oito vizinhos, que são as células adjacentes horizontalmente, verticalmente ou diagonalmente. A cada passo no tempo, ocorrem as seguintes transições:
    - Qualquer célula viva com menos de dois vizinhos vivos morre;
    - Qualquer célula viva com dois ou três vizinhos vivos continua viva para a próxima geração;
    - Qualquer célula viva com mais de três vizinhos vivos morre;
    - Qualquer célula morta com exatamente três vizinhos vivos torna-se uma célula viva.

    A tarefa é fazer um programa de python que implementa o jogo da vida:
    - Uma grade com 100×100 células e condições iniciais randômicas. que dependam de dois parâmetros p_0 e p_1 sendo as probabilidades de iniciar morta ou viva, respetivamente. Testar diferentes valores;
    - Uma grade menor e condições iniciais como as descritas no https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life, de forma a reproduzir alguns padrões clássicos conhecidos como **Still lifes**, **Oscillators** e **Spaceshifts**.
      O programa dever estar estruturado da seguinte forma:

```
# Grid size
N = 100

# Create an initial random grid
p0, p1 = 0.8, 0.2
grid = np.random.choice([0, 1], N*N, p=[p0, p1]).reshape(
    N, N)

def update(frameNum, img, grid):
    # Programar as regras de atualizacao
    .
    .
    .
    plt.title(f"Game of Life - Frame {frameNum}")
    return img,

fig, ax = plt.subplots()
img = ax.imshow(grid, interpolation='nearest')
ani = animation.FuncAnimation(fig, update, fargs=(img,
    grid,))
plt.show()
```

    Explicar que o que cada parte do código faz.

VARIAVEIS E CAMINHADAS ALEATÓRIAS, CÁLCULOS MONTE CARLO

2

2.1 Preludio

Neste capítulo iremos desenvolver alguns cálculos que envolvem variáveis aleatórias. Em alguns casos, a abordagem determinística não é a mais apropriada, por diversos motivos: por exemplo, quando existe incerteza em certas variáveis ou quando a complexidade do problema é tão grande que não conseguimos barrer de maneira exaustiva todos os possíveis valores e combinações que as variaveis envolvidas podem assumir. Nesses casos, a abordagem estocástica torna-se mais conveniente. Vamos apresentar alguns exemplos de cálculos estocásticos e desenvolver códigos em `python` para realizá-los, incluindo, caminhadas aleatórias e cálculo de integrais por métodos Monte Carlo.

Para começar, vamos considerar um exemplo que já foi introduzido:

Exemplo/Exercício 1

Considerar M dados que são jogados, podendo sair números do 1 até o 6. A soma dos valores será

$$S = \sum_{i=1}^{M} d_i$$

em que d_i é o número que saiu em cada um dos dados. Jogar os M dados N vezes e construir um histograma que mostre o comportamento de S. Lembrando, um histograma é um gráfico de frequencia de um certo evento, ou seja, quantas vezes a soma deu M, quantas vezes a soma deu $M + 1$, quantas vezes a soma deu $M + 2$, ..., quantas vezes a soma deu $N \times M$. Para fazer o histograma precissará usar a função

```
counts, bins = np.histogram(s)
plt.stairs(counts, bins)
```

em que `s` será um vetor que possui os resultados das N realizações.

A primeira tarefa será desenvolver um código de `python` que grafica os histrogramas que permitam visualizar como a variável aleatória S está distribuida. Para isto:

- Generalizar o código desenvolvido no capítulo anterior para que o cálculo esteja dentro de uma função, a qual deve receber os valores de N e M;
- Fazer um estudo barrendo diferentes valores de N (p.e., 10^2, 10^3, 10^4, 10^5, 10^6, 10^7) e de M (p.e., 2, 4, 8). Elaborar os histogramas e pesquisar que opções existem para plotar os gráficos.

Este exercício será avaliado!

2.2 Cálculo de integrais por métodos Monte Carlo

Pergunta

Vamos supor que você precisa calcular o valor de π usando o computador. Como você faria sem usar nenhuma fórmula conhecida que envolve series de potências ou coisas do tipo, apenas pode usar cálculo de integrais, mas tem que fazer de conta que você não sabe como calcular essas integrais na mão, apenas tem o computador. Pensar em grupo por 15min e tentar elaborar uma estrategia.

Cálculo do número π

Vamos estimar el valor do número π da seguinte forma: Se consideramos a região quadrada $[-1,1]\times[-1,1]$ e um círculo inscrito, a razão das áreas entre estes será

$$R = \pi/4$$

(ver figura).

Então, podemos propor um cálculo estocástico que consistirá em jogar pontos dentro dessa região quadrada. Alguns desses pontos irão cair dentro do circulo e outros fora. Como é de se esperar, a razão entre a quantidade de pontos que cai dentro e a quantidade total de pontos jogados, deveria tender justamente a razão entre as áreas de círculo e do quadrado ($\pi/4$).

Se denotarmos por N_c e N ao número de pontos dentro do círculo e ao número total de pontos, respectivamente:

$$\text{Prob} = \frac{\pi}{4} = \lim_{N\to\infty} \frac{\#\text{num. pontos no círculo}}{\#\text{num. total de pontos}} = \lim_{N\to\infty} \frac{N_c}{N}$$

Isto leva ao seguinte roteiro que precisamos executar:

1. Gerar (x_i, y_i), formado por dois números x_i e y_i entre -1 e 1, independentes e com probabilidade uniforme.
2. Calcular θ_i, definida como igual a 1 se $x_i^2 + y_i^2 < 1$ e igual a 0 se não. Seja $\Theta = \{\theta_i\}$ a sequência gerada.
3. Tirar a média

$$\hat{\theta} = \frac{1}{N}\sum_{i=1}^{N} \theta_i$$

o qual se materializa no seguinte código de python:

```
# Computation of pi by a stochastic method

N = 10000 # Number of realizations
Nc = 0
for i in range(N):
 x = -1.0 + 2.0*np.random.rand()
 y = -1.0 + 2.0*np.random.rand()
 if (x**2 + y**2 < 1.0):
  Nc = Nc + 1
```

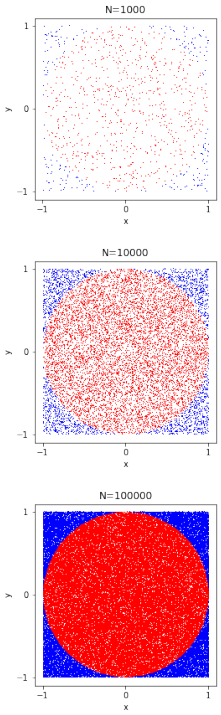


Figure 2.1: Exemplo do cálculo MC.

```
Prob = Nc / N
print('The estimate for pi is:', 4*Prob)
```

Os resultados para diferente número de realizações se mostram na figura ao lado e na tabela seguinte para $4 \times p$ (o que deveria tender a π)[1]:

#	$N = 10^3$	$N = 10^4$	$N = 10^5$	$N = 10^6$	$N = 10^7$
1	3.0800	3.1492	3.1450	3.1418	3.1405
2	3.2320	3.1340	3.1396	3.1400	−−
3	3.1240	3.1412	3.1469	3.1376	−−
4	3.0800	3.1660	3.1488	3.1406	−−

1: Um dos pontos a destacar é que estamos usando uma distribuição uniforme de probabilidades (`np.random.rand()`) para gerar os pontos randômicos, i.e., a probabilidade é a mesma para qualquer número em $(0, 1)$ e os pontos gerados são "independentes" um dos outros. Isto é um ingrediente essencial do método MC.

Exercício 2

- Elaborar um código de cálculo que implementa o método Monte Carlo para calcular o número π. Para isto, encapsular todos os cálculos dentro de uma função que possa ser chamada especificando o número de realizações;
- Elaborar uma tabela de resultados similar à mostrada acima;
- Para cada valor de N, realizar uma média do resultado obtido em cada execução do experimento;
- Plotar o erro em escala `loglog` do resultado (i.e., $|\pi - \pi_{\text{MC}}|$) como função de N. Tirar alguma conclusão. O resultado que irá obter será parecido com o mostrado na figura Figure 2.2.

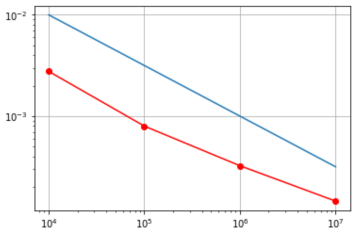


Figure 2.2: Comportamento do erro como função de N. A curva azul mostra a função $N^{-\frac{1}{2}}$ que indica o comportamento teórico esperado. Neste caso foram realizados 20 experimentos e em cada experimento foram jogados até 10^7 pontos.

Exercício 3

Implementar o cálculo de número π usando um método da soma de Riemann (i.e., um método determinístico). Para isto considerar a soma de Riemann

$$S_R = \sum_{i=1}^{N} f(x_i)\,\Delta x$$

em que $f(x) = \sqrt{1 - x^2}$ e $\Delta x = 1/N$, sendo N o número de subintervalos e $x_i = i\,\Delta x$. Claramente, $\lim_{N\to\infty} S_R = \frac{\pi}{4}$.

- Considerar diferentes valores de N e calcular o erro como função de N comparando com o valor de π conhecido.
- Plotar e/ou fazer uma tabela para reportar os resultados.

Exercício 4

Considerar as funções $f(x) = 1 + \frac{1}{2}\,\sin^3(2\,x)$ e $g(x) = 3 + \frac{1}{2}\,\cos^5(3\,x)$ no intervalor $[0, 2\pi]$.

- Plotar as funções.
- Usando uma tabela de integrais ou algum programa para cálculo simbólico (p.e., Mathematica) calcular a área entre as duas curvas.
- Calcular com um método Monte Carlo a área entre as duas curvas e comparar com o item anterior. O programa dever plotar os pontos que caíram fora e dentro da região.

Exercício 5

Considerar as mesmas funções do exercício anterior mas agora considerar que a esquerda e a direita a região está limitada por uma parábola como mostrado na figura ao lado.

A parábola da esquerda passa pelos pontos $[0, 1]$, $[-1, 2.25]$, $[0, 3.5]$ e a parábola de direita passa pelos pontos $[2\,\pi, 1]$, $[2\,\pi+1, 2.25]$, $[2\,\pi, 3.5]$.

- Plotar a região considerada graficando as funções correspondentes.
- Calcular a área de região considerada usando o método Monte Carlo e tabelar o resultado como função do número de pontos sendo jogados. O programa dever plotar os pontos que caíram fora e dentro da região.

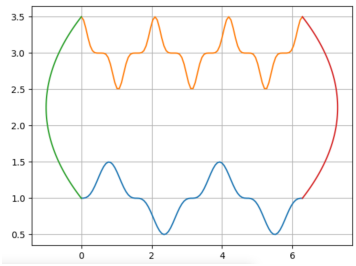


Figure 2.3: Região do exercício 5.

Exercício 6

Calcular o número π usando a técnica Monte Carlo mas agora trabalhando em três dimensões jogando pontos aleatórios dentro do cubo $[-1, 1] \times [-1, 1] \times [-1, 1]$ e considerando a esfera de raio 1 inscrita no cubo:

- Modificar o código original para adaptar-lo ao novo cálculo.
- Fazer uma tabela para plotar os resultados como função do número de pontos N e avaliar o erro.
- Plotar os pontos num gráfico 3D tentando ajustar a transparência dos pontos que ficam fora da esfera para poder enxergar os pontos dentro da esfera.
- Tentar melhorar a eficiência do código vetorizando ele (perguntar ao professor).

Exercício 7

Considerar una moeda "justa" que é jogada no ar N vezes, pudendo sair cara ou cruz. Fazer um algoritmo para estimar a probabilidade de que saim n caras seguidas. Considerar por exemplo $N = 10$ e 20 e $n = 3$ e 5. Será necessário pensar no algoritmo para detetar dentro de um vetor a ocorrência de n caras seguidas. O experimento deverá ser realizado dezenas de milhares de bases para estimar a probabilidade como

$$p = \frac{N_{\text{casos de sucesso}}}{N_{\text{experimentos}}}$$

Não considerar sobreposições, ou seja, com que a sequência se repita apenas 1 vez as n vezes dentro dos N lançamentos, já se considera caso de sucesso.

Exercício 8

Considerar um grafo com N nós, por exemplo, o grafo na sequência:

```
edges = [(0, 1), (1, 3), (1, 2), (2, 4), (2, 6), (6, 5),
      (6, 7), (7, 8), (8, 9)]
```

(i) Considerar uma partícula que entra no nó 0 e precisa sair pelo nó $N-1$.
(ii) Um nó que já foi visitado pela partícula não pode mais ser visitado.
(iii) A cada passo, a partícula pode pular de nó atual para os vizinhos que ainda não foram visitados, com probabilidade uniforme, i.e., se um nó está conectado com outros 3 nós que ainda não foram visitados, a probabilidade de cair em qualquer um deles é $\frac{1}{3}$.

- Realizar M experimentos de caminhada da partícula no grafo para estimar a probabilidade da partícula conseguir chegar no nó $N-1$.
- Registrar os caminhos que a partícula percorreu e imprimir os resultados na tela.
- Inventar outros grafos mais complicados e rodar o código novamente. Interpretar os resultados.

2.3 Caminhadas aleatórias

Vamos supor uma grade quadrada em $\mathbb{R}^2$ e uma partícula na posição inicial $\mathbf{R}(0)$. Queremos estudar a evolução dessa partícula quando a mesma realiza uma caminhada aleatória, i.e., se $R(t)$ representa a sua posição no momento t, então a sua posição no tempo $t+1$ será dada por:

$$\mathbf{R}(t+1) = \mathbf{R}(t) + \boldsymbol{\xi}_t = \mathbf{R}(0) + \sum_{i=1}^{t} \boldsymbol{\xi}_i$$

em que o vetor $\boldsymbol{\xi}_t$ pode tomar um dos 4 valores na sequência:

$$(a,0),\ \ (0,a),\ \ (-a,0),\ \ (0,-a)$$

cada um destes com uma probabilidade uniforme de $\frac{1}{4}$. No geral, a trajetória desta partícula viajante (ou caminhante) terá a forma mostrada na figura 2.4

Exercício 9

Implementar um código que realiza caminhadas aleatórias em $2D$.

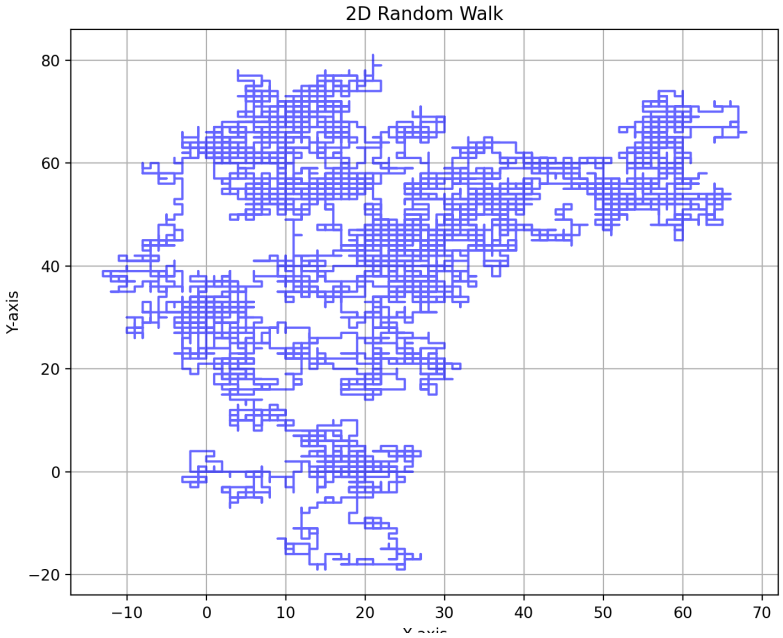
- Plotar o resultado para várias realizações da caminhada e considerando $N = 10^3,\ 10^4, 10^5, 10^6$ passos.
- Calcular o distância quadrática média percorrida, i.e.,

$$d_m^2(t) = \langle (\mathbf{R}(t) - \mathbf{R}(0)) \cdot (\mathbf{R}(t) - \mathbf{R}(0)) \rangle$$

  como função de t e fazendo um promédio sobre M realizações, tomando por exemplo $M = 1000$ e $M = 10000$ e $N = 10^4$ passos.
- Estender o código para 3D.

Exercício 10 (opcional)

Estudo de **agregados** fractais:



- Considerar uma grade de 400×400 células de tamanho unitário.
- Colocar uma particula no centro da grade.
- Tomar n particulas em posições aleatórias, que iniciem a uma distância $D > 180$ do centro da grade
- Executar as caminhadas aleatórias destas partículas.
- Cada uma dessas caminhadas se dá por finalizada se a partícula sai da grade ou se fica "grudada" a uma particula que já faz parte do **agregado**, i.e., se ela chega numa posição que tem por vizinho alguma partícula já grudada ao sistema.
- O resultado deveria se parecer com a Figure 2.5.

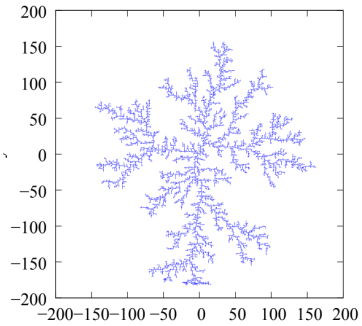


Figure 2.5: Agregado por difusão.

PROCESSOS ITERATIVOS E RELAXAÇÕES 3

3.1 Preludio

Neste capitulo o foco será dado a diversos processos iterativos que aparecem certos problemas da física. A ideia de um processo iterativo, é construir uma sequência de aproximações para um problema, i.e.,

$$\{a_0, a_1, \ldots, a_k, \ldots\}$$

em que a_0 é a condição inicial, a qual é dada ou conhecida. A quantidade a pode representar alguma grandeza física, tal como a temperatura num conjunto de pontos, a posição de uma partícula, a solução de um sistema de equações, etc. Este tipo de cálculos são ideias para serem programados no computador de maneira eficiente como veremos neste capítulo. Como é de se esperar, as estruturas de repetição que temos aprendido são essenciais para implementar estes cálculos no computador, i.e., of `for` e o `while`.

3.2 Exemplos elementares de processos iterativos

Exemplos:

(a) **Sequência de Fibonacci:** A famosa sequência de Fibonacci é dada por:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 1 \\ a_k &= a_{k-1} + a_{k-2}, \quad k = 2, 3, \ldots \end{aligned}$$

Neste caso particular, os dois primeiros elementos da sequência são dados e o resto são calculados usando a fórmula anterior.

(b) **Mapeo logístico:** Este exemplo que já foi estudado no primeiro capítulo é dado por:

$$x_n = a\, x_{n-1}\,(1 - x_{n-1}), \quad n = 1, 2, \ldots, N$$

em que a é um parâmetro dado e x_0 precisa ser escolhido.

(c) **Iteração de ponto fixo:** Dada uma função $\varphi(x)$, a iteração de ponto fixo é definida por

$$x_{k+1} = \varphi(x_k)$$

com x_0 dado. Diz-se que $\bar{x}$ é ponto fixo da função φ se $\bar{x} = \varphi(\bar{x})$ e a iteração de ponto fixo diz-se convergente se:

$$\lim_{k\to\infty} x_k = \bar{x}$$

(d) **Iteração de Jacobi:** Considerar o sistema de equações:

$$\begin{aligned} 3\,x - y &= 2 \\ -2\,x + 4\,y &= 1 \end{aligned}$$

O processo iterativo de Jacobi, parte de uma condição inicial $[x_0, y_0]$ e a partir desta atualiza a cada iteração seguindo a regra:

$$\begin{aligned} x_{k+1} &= \frac{1}{3}(2 + y_k) \\ y_{k+1} &= \frac{1}{4}(1 + 2\,x_k) \end{aligned}$$

Como pode-se ver, a ideia é que em cada equação se atualiza o valor de uma das incógnitas com os valores das outras incógnitas na iteração anterior. O processo iterativo deve ser executado até que os valores de x_k e y_k estão suficientemente próximo da solução do sistema, i.e., para k suficientemente grande, se denotarmos por $(\bar{x}, \bar{y})$ a solução do sistema:

$$|\bar{x} - x_k| < \varepsilon, \quad |\bar{y} - y_k| < \varepsilon$$

com ε pequeno. Em analise numérica pode-se demonstrar que para certos tipos de sistemas, independentemente da condição inicial x_0, y_0, esse processo iterativo é convergente.

(e) **Método de Newton:** Dada uma função $f(x)$ cujos zeros querem ser achados, o processo iterativo:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

dependendo da função f e do valor inicial x_0, o processo iterativo pode converger a um valor $\bar{x}$ tal que $f(\bar{x}) = 0$ (i.e., uma raiz de f). O processo dever ser executado até que o valor de x_k está suficientemente próximo da solução da equação. Em analise numérica, pode-se demonstrar que dependendo da função f e do valor escolhido para x_0, esse processo iterativo é convergente.

(f) **Método das potencias:** Dada uma matriz $A \in \mathbb{R}^{n\times n}$ e um vetor $\mathbf{x}_0 \in \mathbb{R}^n$ de norma unitaria, no método das potencias calcula-se um novo vetor $\mathbf{x}_{k+1}$ e um escalar λ_{k+1} a partir do vetor anterior seguindo a regra

$$\mathbf{y}_{k+1} = A\,\mathbf{x}_k, \quad \mathbf{x}_{k+1} = \frac{\mathbf{y}_{k+1}}{\|\mathbf{y}_{k+1}\|}, \quad \lambda_{k+1} = \mathbf{x}_{k+1}^{\mathsf{T}}\,A\,\mathbf{x}_{k+1}$$

Notar que $\mathbf{y}_{k+1} = A\,\mathbf{x}_k = A^2\,\mathbf{x}_{k-1} = \cdots = A^{k+1}\,\mathbf{x}_0$. Dai o nome do método. Dependendo da matriz A, o método converge ao autovetor $\bar{\mathbf{x}}$ correspondente ao autovalor dominante $\bar{\lambda}$ da matriz (i.e., o autovalor de maior módulo).

Exercícios para a avaliação

1. Implementar em `python` o exemplo (a) da sequência de Fi-

bonacci. Programar-lo armazenando os resultados num vetor e programar-lo usando o conceito de recorrência, i.e., onde a função chama-se a si mesmo tantas vezes quanto necessário para calcular os elementos da sequência.

2. Fazer um programa de `python` que executa o processo de iteração de ponto fixo do exemplo (c) para a função $\varphi(x) = 1/\sqrt{x}$ e graficar o processo iterativo como mostrado na figura ao lado. Notar que a condição inicial escolhida no exemplo é $x_0 = 0.75$.

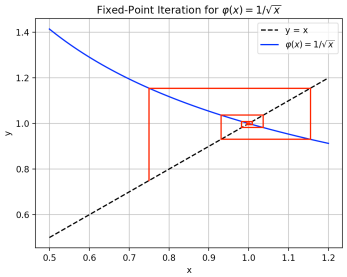


Figure 3.1: Exemplo de iteração de ponto fixo.

3. Programar o processo iterativo de Jacobi do exemplo (d) e verificar quantas iterações são necessárias para atingir um erro $\varepsilon = 10^{-2}, 10^{-3}, 10^{-4}, \ldots, 10^{-8}$. Graficar o número de iterações como função do erro.
4. Programar o processo iterativo de Jacobi para o sistema de 3×3 da sequência:

$$\begin{bmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

5. Programar o método de Jacobi para um sistema geral de equações em que a matriz $A \in \mathbb{R}^{n \times n}$ e o vetor de lado direito $\mathbf{b} \in \mathbb{R}^n$ são dados, i.e.,

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ . & & & \\ . & & & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ b_n \end{bmatrix}$$

   Notar que neste sistema não conhecemos facilmente a solução, justamente, a ideia do método proposto é achar ela, por tanto não podemos avaliar a grandeza $|\bar{x}^i - x_k^i|$, onde o supraíndice refere-se à i-éssima incógnita e o subíndice à k-éssima iteração. Então, nesse caso como podemos saber em que momento parar o processo iterativo? Pense alguma alternativa.
6. Implementar em `python` o processo iterativo de Newton do exemplo (e), para encontrar as duas raizes no intervalo [0,2] da função:

$$f(x) = x\, e^{-x^2} - 0.1$$

   Primeiro plotar a função no intervalo [-1,2]. Baseado no gráfico da função escolher pontos iniciais x_0 que estejam próximos das raizes procuradas. Escolher um erro ε pequeno e determinar quantas iterações são necessárias para a convergência do método.
7. Implementar o método das potencias do exemplo (f) na matriz

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

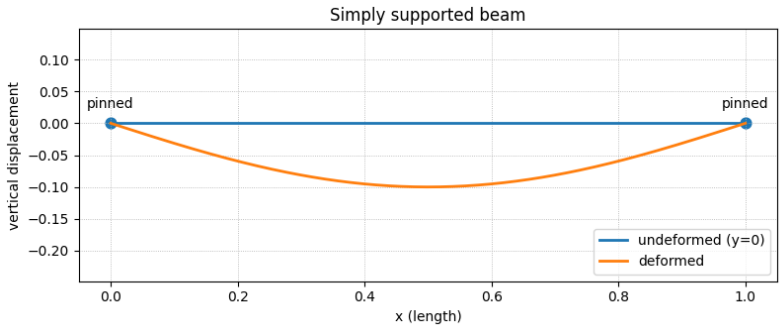
e na matriz

$$A = \text{diag}(a_1, a_2, \ldots, a_n)$$

sendo a_is números randômicos uniformemente distribuídos entre 0 e 1.

3.3 Forma de equilibrio de uma corda elástica

Vamos considerar uma corda em tensão como mostrado na figura.



- A tensão da corda será denotada por τ [N] e a mesma é conhecida;
- Para efeitos de modelagem, consideramos que a corda é discretizada em N segmentos de comprimento ℓ_0, sendo o comprimento total da corda denotado por L, então $L = N\,\ell_0$;
- Os extremos da corda estão presos à parede, isto significa que o ponto $x = 0$ e $x = L$ não podem se deslocar;
- O massa total da corda é dada por:

$$M = \int_0^L \rho(x)\,dx$$

  em que a densidade $\rho(x)$ [kg/m] é uma função conhecida.
- Para efeitos de modelagem consideramos que a massa total da corda M é concentrada em $n = N - 1$ partículas de massa m_i equidistantes localizadas a distância ℓ_0 entre elas, sendo a massa da i-éssima partícula

$$m_i = \int_{(i+1)\,\ell_0-\frac{\ell_0}{2}}^{(i+1)\,\ell_0+\frac{\ell_0}{2}} \rho(x)\,dx, \quad i = 1, 2, \ldots, n-1$$

  e

$$m_0 = \int_0^{\frac{3}{2}\,\ell_0} \rho(x)\,dx, \quad m_{n-1} = \int_{L-\frac{3}{2}\,\ell_0}^{L} \rho(x)\,dx,$$

- Em cada massa está aplicada a força para baixo devido a gravidade, i.e.,

$$p_i = -m_i\,g$$

- Em certas massas pode estar aplicada alguma força adicional f_i
- A corda encontra-se em equilíbrio estático devido ao balanço das forças;
- Fazendo um balanço de forças na direção vertical, podemos escrever que:

$$\mathscr{F}_i^y = 0, \quad i = 0, \ldots, n-1$$

  em que

$$\mathscr{F}_i^y = -m_i\,g + f_i + F_i^\sigma$$

- Para calcular a força devido a tensão da corda usamos que:

$$F_i^\sigma = -\tau\,(\sin\theta_L^i + \sin\theta_R^i)$$

sendo que para ângulos **pequenos**, podemos usar a seguinte aproximação para $i = 1, \ldots, n-1$

$$\sin\theta_L^i \approx \tan\theta_L^i = \frac{y_i - y_{i-1}}{\ell_0}, \quad \sin\theta_R^i \approx \tan\theta_R^i = \frac{y_i - y_{i+1}}{\ell_0},$$

Aqui já temos acertado os sinais de tudo para que resulte o sistema da forma correta. Estas fórmulas servem para todas as massas se definirmos

$$y_{-1} = y_n = 0$$

pois os extremos da cordas não podem se deslocar.

- Colocando tudo junto, para cada massa temos a equação de equilíbrio:

$$-m_i\,g + f_i - \frac{\tau}{\ell_0}\,(-y_{i-1} + 2\,y_i - y_{i+1}) = 0 \tag{3.1}$$

Notar que no caso de densidade constante, podemos escrever para pontos que não sejam o primeiro ou o último:

$$-\rho\ell_0\,g + f_i - \frac{\tau}{\ell_0}\,(-y_{i-1} + 2\,y_i - y_{i+1}) = 0 \tag{3.2}$$

Vale mencionar que uma forma mais correta de formular as equações seria escrever o problema como uma EDO

$$-\tau\frac{d^2y}{dx^2} = -\rho(x)g + f(x)$$

e aplicar um método de discretização tal como o método das diferenças finitas para aproximar a derivada segunda que aparece no lado esquerdo, porém isto é estudado em cursos mais avançados.

Mini projeto

Considerar uma corda de comprimento $L = 1$m, tensão $\sigma = 2$ N:

- Considerar a corda particionada em N intervalos e duas funções de densidade

  (i) $\rho(x) = 1$ kg/m
  (ii) $\rho(x) = \frac{1}{2}(1 + e^{-100\,(x-\frac{1}{2})^2})$ kg/m

  Fazer uma função de `python` para calcular a massa das N particulas em ambos casos e para valores de $N = 10, 20, 40, 80$.

- Implementar um método iterativo pelo qual é encontrada a posição de equilíbrio das massas usando iterações de Jacobi, i.e., o posição da i-éssima massa é achada a partir da posição das massas vizinhas na iteração prévia.

- Implementar o método iterativo, conhecido como esquema de Gauss-Seidel. Neste método, para achar a posição de equilíbrio da i-éssima massa, se formos avançando em orden progressiva,

i.e., $0, 1, 2, \ldots$, emprega-se a posição da massa $i + 1$ na iteração prévia (pois ainda esta não tem sido calculada) e a a posição da massa $i - 1$ na iteração atual, pois ela sim já tem sido calculada.

Nos dois itens anteriores considerar $N = 10, 20, 40, 80$, $g = 9.8\text{m/s}^2$ e os dois casos de densidade dados anteriormente e plotar a forma de equilíbrio da corda em cada caso.

- Para o caso com $\rho = 1$ e $N = 80$, aplicar uma força vertical para baixo f_{39} na massa $i = 39$ de valor variável e plotar o deslocamento sofrido pelo ponto como função de f_{39}.
- Repetir o anterior mas aplicando ao mesmo tempo uma força para cima na massa $i = 17$ e uma força para baixo na massa $i = 30$. Plotar a forma de equilíbrio.
- Agora considerar:
  - A corda encontra-se imersa num fluido com densidade $\rho_f = 1000\text{kg/m}^3$ (a.k.a. água)
  - Vamos supor que o diámetro da corda é uma função de x dada por:
    $$\phi(x) = 2\,(1 + e^{-200\,(x-\frac{1}{2})^2})$$
  - Considerar que a densidade volúmetrica $\bar{\rho}$ do material da corda é 900 kg/m^3.
  - Considerar as f_is iguais a zero.
  - Incluir a força de empuxo que o fluido faz sobre a corda, calculando o volume da corda e distribuindo ele sobre as partículas:
    $$f_i^b = \rho_f v_i g$$
  - Calcular a forma de equilibrio da corda e plotar.

TÓPICOS VARIADOS DE PROGRAMAÇÃO EM python

4

4.1 Preludio

Neste capítulo iremos estudar algumas ferramentas adicionais para completar a formação nas técnicas de programação em `python`, incluindo, algumas noções sobre processamento de imagens, algoritmos para procura e ordenação de arrays e finalmente algumas funcionalidades interessantes para gerar gráficos em 2D e 3D.

4.2 Processamento de imagens

Uma imagem não é outra coisa que uma lista de matrizes. Por exemplo para uma imagem RGB teriamos uma lista de 3 matrizes. Considerar o código

```
import matplotlib.image as mpimg
img = mpimg.imread('stinkbug.png')
plt.imshow(img)
```

Figure 4.1: Imagem em tons de cinza.

(o arquivo `stinkbug.png` está disponível no Tidia).

O objeto `img` é um array de 3 índices, como pode-se certificar vendo a dimensão do mesmo:

```
> print(img.shape)
(375, 500, 3)
```

Cada matriz da lista representa um canal e os valores dentro da matriz a intensidade de cada um dos 375×500 **pixels** da imagem. Se imprimirmos o array veremos, neste caso particular, que todas as matrizes são muito parecidas pois esta é uma imagem em tons de cinza.

```
array([[[0.40784314, 0.40784314, 0.40784314],
        [0.40784314, 0.40784314, 0.40784314],
        [0.40784314, 0.40784314, 0.40784314],
        ...,
        [0.42745098, 0.42745098, 0.42745098],
        [0.42745098, 0.42745098, 0.42745098],
        [0.42745098, 0.42745098, 0.42745098]],

       [[0.4117647 , 0.4117647 , 0.4117647 ],
        [0.4117647 , 0.4117647 , 0.4117647 ],
        [0.4117647 , 0.4117647 , 0.4117647 ],
        ...,
        [0.42745098, 0.42745098, 0.42745098],
        [0.42745098, 0.42745098, 0.42745098],
```

```
 [0.42745098, 0.42745098, 0.42745098]],

[[0.41960785, 0.41960785, 0.41960785],
 [0.41568628, 0.41568628, 0.41568628],
 [0.41568628, 0.41568628, 0.41568628],
 ...,
 [0.43137255, 0.43137255, 0.43137255],
 [0.43137255, 0.43137255, 0.43137255],
 [0.43137255, 0.43137255, 0.43137255]],
```

Para manipular a imagem por simplicidade podemos pegar apenas um canal dela, p.e.

```
A = img[:, :, 0]
```

e realizar operações sobre esta matriz. É isso que será desenvolvido justamente nos seguintes exercícios.

Exercícios

1. Considerar a imagem do exemplo anterior. Visualizar a imagem usando o comando `plt.imshow`. Visualizar a imagem original `img` e visualizar o canal `A` considerando diferentes mapas de cores:

   ```
   plt.imshow(A)
   plt.imshow(A, cmap="hot")
   plt.imshow(A, cmap="'nipy_spectral'")
   ```

   O resultado deveria ser algo assim:

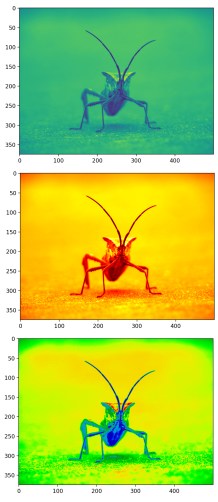


   Figure 4.2: Visualização esperada da imagem.

2. Para a mesma imagem do exemplo, modificar a mesma para que apareçam manchas circulares de cor branco e de cor preto em posições aleatórias.
3. Programar uma função `Suaviza` para processar imagens. Para isto será necessário carregar uma imagem
   - Fazer uma função de suavizado em que a matriz resultante `B`, em cada pixel (ou célula), tenha a média dos valores das células correspondentes e das células adjacentes em horizontal e em vertical da matriz de entrada `A`, i.e., pensando a imagem como uma matriz

     $$B_{ij} = \frac{1}{5}(A_{i,j} + A_{i+1,j} + A_{i-1,j} + A_{i,j+1} + A_{i,j-1})$$

     Aplicar o algoritmo repetidas vezes e observar o resultado como função do número de vezes que ele é aplicado.
   - Fazer outra versão em que matriz resultante, em cada pixel, tenha a média dos valores das células correspondentes, das células adjacentes em horizontal, em vertical e em diagonal.

Tomar providências para o tratamento dos pixels que estão na borda e aplicar as funções repetidas vezes em alguma imagem que possua tons de cinza. Testar o algoritmo na imagem do exemplo anterior ou em outra imagem que achar.

4. Considerar a imagem de microscopio disponível no tidia no arquivo `concrete.jpg` a qual corresponde a uma amostra de concreto usado na construção civil.

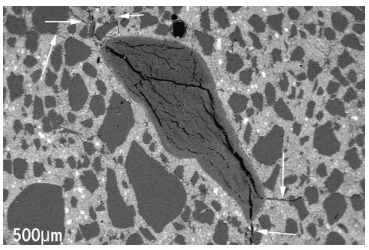


Figure 4.3: Imagem de uma amostra de concreto usada na construção civil.

   - Para carregar a mesma pode considerar a parte inicial do exercício anterior e ficar apenas com um canal. Estude a matriz que resulta.
   - Recortar uma janela da imagem que esteja livre de anotações. Considere diferentes tamanhos de janela.
   - Apartir da imagem recortada no item anterior, elaborar um algoritmo que processe a imagem para estimar a fração de agregados, i.e., as partes em cinza mais escuro. Para isto, precisará normalizar a imagem e determinar um limiar de ton de cinza a partir do qual o pixel pode ser considerado agregado. Para isto, considerar o código:

     ```
     img = (recortada < limiar)*1.0
     ```

     Analisar os resultados como função do valor tomado para `limiar`.

4.3 Mais sobre gráficação e animação

4.3.1 Plotagem de funções de 2 variáveis

Para completar o nosso conhecimento de biblioteca `matplotlib` precisamos estudar algumas funções para realizar gráficos de funções. Específicamente, queremos gráficar:

- Superfícies em 3D;
- Curvas de nível constante;
- Campos vetoriais;

No execício na sequência a ideia é pesquisar no site oficial do `matplotlib` (`https://matplotlib.org/`) para conseguir reproduzir os gráficos mostrados.

Exercícios

Considerar a função

$$f(x, y) = \cos(x)\,\cos(y)$$

- Usando a função `plt.plot_surface`, plotar f como superfície em 3D, considerando diferentes pontos de vista;
- Usando a função `plt.contourf`, plotar gráficos com curvas de nível da função usando diferentes escalas de cores;

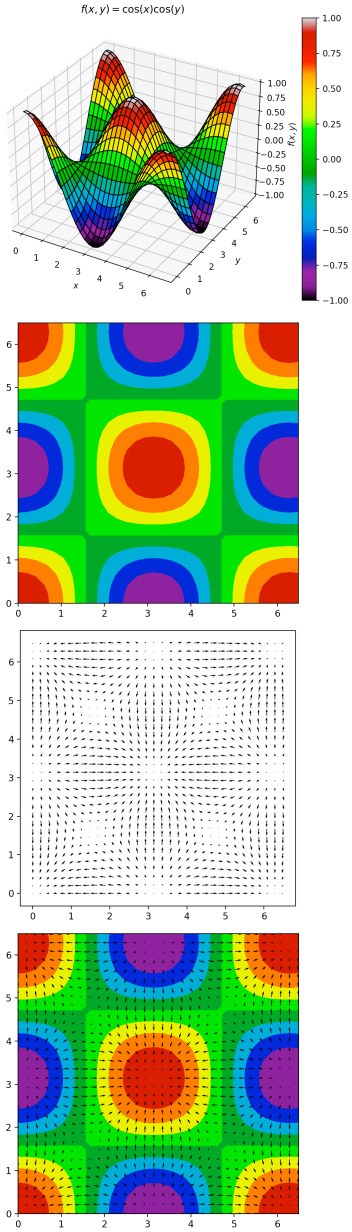


Figure 4.4: Gráficos da função f.

- Usando a função `plt.quiver`, plotar o gradiente da função $\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]^\intercal$. Superpor o gráfico com as curvas de nível constante e verificar visualmente que estas são ortogonais ao gradiente.

Os gráficos tem que ser similares aos mostrados nas figuras.

4.3.2 Um sistema dinâmico

Considerar dois corpos de massas m_1 e m_2, interagindo gravitacionalmente de acordo com a fórmula:

$$\mathbf{F}_{2\to1}(\mathbf{r}_1(t), \mathbf{r}_2(t)) = -\frac{\gamma m_1 m_2}{\|\mathbf{r}_{12}\|^3}\mathbf{r}_{12} = -\mathbf{F}_{1\to2}(\mathbf{r}_1(t), \mathbf{r}_2(t))$$

onde $\mathbf{r}_{12} = \mathbf{r}_2 - \mathbf{r}_1$, sendo $\mathbf{r}_i$ a posição do i-éssimo corpo. Sendo uma força central, o movimento está restrito ao plano. Um esquema númerico para calcular a evolução dos corpos, partindo da condição inicial dada

$$[\mathbf{r}_i(0), \mathbf{v}_i(0)] = \left[\mathbf{r}_i^0, \mathbf{v}_i^0\right]$$

é dado no pseudo-codigo da sequência:

Método de avanço no tempo

Dada a condição inicial $\left[\mathbf{r}_i^0, \mathbf{v}_i^0\right]$ e o passo de tempo Δt

Inicializar $t^0 = 0$, $n = 0$

Enquanto $t^n < T_{\text{fin}}$

1. Calcular a aceleração: $\mathbf{a}_i^n = \frac{\mathbf{F}_{j\to i}(\mathbf{r}_1^n, \mathbf{r}_2^n)}{m_i},\ j \neq i,\ i = 1, 2$
2. Atualizar a posição: $\mathbf{r}_i^{n+1} = \mathbf{r}_i^n + \mathbf{v}_i^n \Delta t + \frac{1}{2}\mathbf{a}_i^n \Delta t^2$
3. Atualizar a aceleração: $\mathbf{a}_i^{n+1} = \frac{\mathbf{F}_{j\to i}(\mathbf{r}_1^{n+1}, \mathbf{r}_2^{n+1})}{m_i},\ j \neq i$
4. Atualizar a velocidade: $\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \frac{1}{2}\left(\mathbf{a}_i^n + \mathbf{a}_i^{n+1}\right)\Delta t$
5. Guardar t^n, $\left[\mathbf{r}_i^n, \mathbf{v}_i^n\right]$
6. Incrementar o tempo $t^{n+1} = t^n + \Delta t$
7. Incrementar n

Fim

Este esquema chama-se Método Verlet-Velocidade e é muito usado na física computacional. No exercício a seguir o método é implementado para estudar as trajetórias de dois corpos.

Exercícios

Na sequência considerar:

- $\gamma = 1$, $\Delta t = 0.01$, $m_1 = 1$, $N_{\text{steps}} = 2000$
- $\left[\mathbf{r}_1^0, \mathbf{v}_1^0\right] = [[0,\ 0], [0,\ 0]]$,
- $\left[\mathbf{r}_2^0, \mathbf{v}_2^0\right] = [[1,\ 0], [0,\ 1]]$

1. Implementar o método Verlet-Velocidade numa função que retorne as posições e velocidades dos corpos como função do tempo.
2. Implementar uma função que salve o histórico completo num arquivo de texto de tipo ascii. Para isto considerar as duas variantes na sequência:
   - Salvar posições e velocidades a medida que estão sendo calculadas:

```
.
.
tn = 0
evolfile = open('evol.txt', 'w+')
n = 0
while(tn < Tfin):
    .
    # Do your calculations
    .
    # Save the current time step
    evolfile.write("%lf %lf %lf %lf %lf %lf ...\n"
            %(tn, r1[0], r1[1], v1[0], v1[1], r2[0],
    ...))

    tn = tn + Dt
    n = n + 1


evolfile.close()
```

   - Ir guardando o histórico de posições e velocidades num `array` e no final salvar ele num arquivo de uma vez:

```
.
.
tn = 0
n = 0
while(tn < Tfin):
    .
    # Do your calculations
    .
    # Store the current time step
    solution[n, :] = tn, r1[0], r1[1], v1[0], v1[1],
    r2[0] ...

    tn = tn + Dt
    n = n + 1

# Save everything at once
np.savetxt(filename, solution)
```

3. Implementar uma função que carregue o arquivo ascii do ponto anterior e plota as trajetórias dos corpos e a energia cinética total do sistema como função do tempo, i.e.,

$$\mathcal{K}(t) = \frac{1}{2}m_1\|\mathbf{v}_1\|^2 + \frac{1}{2}m_2\|\mathbf{v}_2\|^2$$

Para isto, usar a função `np.loadtxt` de `numpy`, a qual recebe como

argumento o nome do arquivo que quer ser carregado.

Para realizar as simulações considerar $m_2 = 0.001$, $m_2 = 0.01$, $m_2 = 0.1$, $m_2 = 1$, $m_2 = 2$.

4. (**Opcional**) Realizar uma animação para ilustrar o movimento dos corpos.

4.3.3 Algoritmo de *flooding*

A ideia é simular o processo de alagamento dentro de uma grade quadrada com $N \times N$ células. No centro da grade temos uma fonte de agua, a qual se espalha ao resto das células da grade. Dentro da grade temos então as células em três possíveis estados:

- Célula não alagada, com valor: 0
- Célula alagada, com valor: 1
- Célula obstáculo, com valor: 2

As regras de atualização, de um passo para o seguinte, são simples: "Se uma célula estiver alagada, então os seus vizinhos (acima, à direita, à esquerda e embaixo) ficam alagados, exceito se a célula for um obstáculo"

Exercícios

Implementar um algoritmo de alagamento numa grade de $N \times N$ células e executa uma animação a medida que o algoritmo avança. Para isto, considerar:

- Uma função `init` que inicializa a grade com um ponto central que encontra-se no estado 1 e coloca diferentes configurações de obtáculos que você achar interessante;
- Uma função `update` que, dada a grade no tempo atual, retorna a nova grade que resulta de aplicar as regras ditas acima;
- Considerar diferentes valores de N;
- Usar a função `FuncAnimation` e experimentar com os valores dos parâmetros `frames` e `interval` para ver o que acontece.

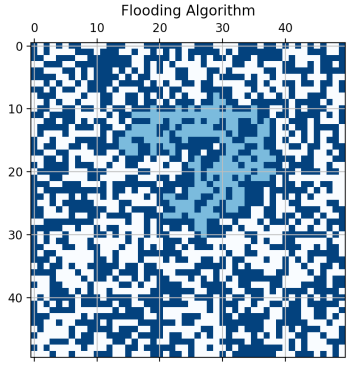


Figure 4.5: Resultado típico do algoritmo de flooding.

4.4 Ordenamento e procura em arrays

4.4.1 Ordenamento de arrays

Em computação é essencial contar com algoritmos de ordenamento que sejam eficientes. Nesta seção iremos considerar três algoritmos que servem para ordenar na ordem crescente os elementos de um vetor:

- Ordenamento de tipo bolha (**Bubble sort**)
- Ordenamento por seleção (**Selection sort**)
- Ordenamento rápido (**Quick sort**)

que estão dados na sequência:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

     return arr
```

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Exercício

1. Explicar como que cada algoritmo funciona.
2. Criar um exemplo simples e mostrar o funcionamento passo a passo de cada algoritmo.
3. Criar vetores de inteiros possuindo N componentes e aplicar os algoritmos. Para isto, considerar $N = 100, 200, 400, 800, 1600$ e medir o tempo consumido em cada caso. Plotar o tempo como função de N usando a escala `loglog` para todos os algoritmos. Tirar conclusões.

Alphabetical Index