Universidade de Brasília
Faculdade de Tecnologia

Tratamento estatístico e interpretação dos resultados obtidos a partir da medição de velocidades turbulentas com anemometria de fio quente

Felipe Andrade

TRABALHO DE LABORATÓRIO

INTRODUÇÃO À TURBULÊNCIA

Brasília

2022

Universidade de Brasília
Faculdade de Tecnologia

Tratamento estatístico e interpretação dos resultados obtidos a partir da medição de velocidades turbulentas com anemometria de fio quente

Felipe Andrade

Trabalho de Laboratório da Disciplina de Introdução à Turbulência

Orientador: Prof. Dr. José Luis Alves da Fontoura

Brasília
2022

Lista de ilustrações

Figura 1 – Descrição geométrica do sistema de coordenadas no túnel de vento . . . 6
Figura 2 – Gráfico Velocidades Instantâneas lre1 22
Figura 3 – Gráfico Velocidades Instantâneas lre2 22
Figura 4 – Gráfico Velocidades Instantâneas lre3 22
Figura 5 – Gráfico Velocidades Instantâneas lre4 22
Figura 6 – Gráfico Velocidades Instantâneas lre5 22
Figura 7 – Gráfico Densidade de Probabilidade lre1 23
Figura 8 – Gráfico Densidade de Probabilidade lre2 23
Figura 9 – Gráfico Densidade de Probabilidade lre3 23
Figura 10 – Gráfico Densidade de Probabilidade lre4 23
Figura 11 – Gráfico Densidade de Probabilidade lre4 23
Figura 12 – Gráfico Velocidades Instantâneas hre1 24
Figura 13 – Gráfico Velocidades Instantâneas hre2 24
Figura 14 – Gráfico Velocidades Instantâneas hre3 24
Figura 15 – Gráfico Velocidades Instantâneas hre4 24
Figura 16 – Gráfico Velocidades Instantâneas hre5 24
Figura 17 – Gráfico Densidade de Probabilidade hre1 25
Figura 18 – Gráfico Densidade de Probabilidade hre2 25
Figura 19 – Gráfico Densidade de Probabilidade hre3 25
Figura 20 – Gráfico Densidade de Probabilidade hre4 25
Figura 21 – Gráfico Densidade de Probabilidade hre4 25
Figura 22 – Gráfico Velocidades Médias hre-prob 26
Figura 23 – Gráfico Densidade de Probabilidade hre-prob 26
Figura 24 – Gráfico Perfil de Velocidade Média 27
Figura 25 – Gráfico Perfil de Intensidade de Turbulência 27
Figura 26 – Gráfico Perfil de Velocidade Média 27
Figura 27 – Gráfico Perfil de Intensidade de Turbulência 27

Lista de tabelas

Tabela 1 – Tabela de Resultados Para lre 20
Tabela 2 – Tabela de Covariâncias e Correlações Para lre 20
Tabela 3 – Tabela de Resultados Para hre 21
Tabela 4 – Tabela de Covariâncias e Correlações Para hre 21
Tabela 5 – Tabela de Resultados Para hre-prob 21

Sumário

1	**OBJETIVOS**	**6**
1.1	**Extração e organização de dados**	**7**
1.2	**Função Densidade de Probabilidade**	**9**
1.2.1	Função de Densidae de Probabilidade Gaussiana	10
1.3	**Médias**	**10**
1.3.1	Média Temporal	11
1.3.2	Média Estatística	12
1.3.3	Média Espacial	12
1.4	**Flutuações**	**13**
1.5	**Energia Cinética de Turbulência por Unidade de Massa**	**13**
1.6	**Intensidade de Turbulência**	**13**
1.7	**Momentos de Ordem Superior**	**14**
1.7.1	Variância	14
1.7.2	Desvio Padrão	14
1.7.3	Coeficiente de Assimetria (*Skewness*)	15
1.7.4	Coeficiente de Achatamento (*Kurtosis*)	15
1.8	**Covariância e Correlação**	**16**
1.8.1	Covariânia	16
1.8.2	Coeficiente de Correlação	17
2	**METODOLOGIA NUMÉRICA E RESULTADOS**	**18**
2.1	**Metodologia Numérica**	**18**
2.2	**Resultados**	**20**
2.2.1	Dados lre	20
2.2.2	Dados hre	20
2.2.3	Resultados hre-prob	21
2.2.4	Perfil Montante	21
3	**CONCLUSÃO**	**28**
3.0.1	Pósfácio	28
	REFERÊNCIAS	**29**

APÊNDICES 30

APÊNDICE A – CÓDIGO FONTE 31
A.1 Definição de Classes 31
A.2 Definição de Funções de Cálculos Numéricos 37
A.3 Código para o tratamento de dados 40
A.4 Código para rodar o programa 41
A.5 Código para realizar a plotagem dos gráficos 43

1 Objetivos

O objetivo do presente trabalho compreende o estudo de escoamento turbulento no entorno de um corpo robundo. Para isto foram realizadas medições com um anemômetro de fio quente em um túnel de vento, os anemômetros foram posicionados em 5 (cinco) pontos do escoamento: com relação ao eixo y os pontos foram colocados em sua origem ou seja, na mediatriz compreendida na face yz do túnel de vento; com relação ao eixo z todos os pontos foram colocados a uma altura de $25mm$; com relação ao eixo x podemos definir o seguinte vetor com as posições de cada um dos anemômetros:

$$P_X = (-75, 50, 75, 100, 125)[mm]$$

1. 5 medições espaciais para um regime de escoamento caracterizado por um baixo número de Reynolds
2. 5 medições espaciais para um regime de escoamento caracterizado por um alto número de Reynolds
3. 25 medições no ponto 5 para um regime de escoamento caracterizado por um alto número de Reynolds
4. 18 medições no ponto 1 para o levantamento dos perfis de velocidade em um escoamento com alto número de Reynolds
5. 18 medições no ponto 3 para o levantamento dos perfis de velocidade em um escoamento com alto número de Reynolds

O sistema de coordeandas do túnel de vento são esauematizadas na figura(1).

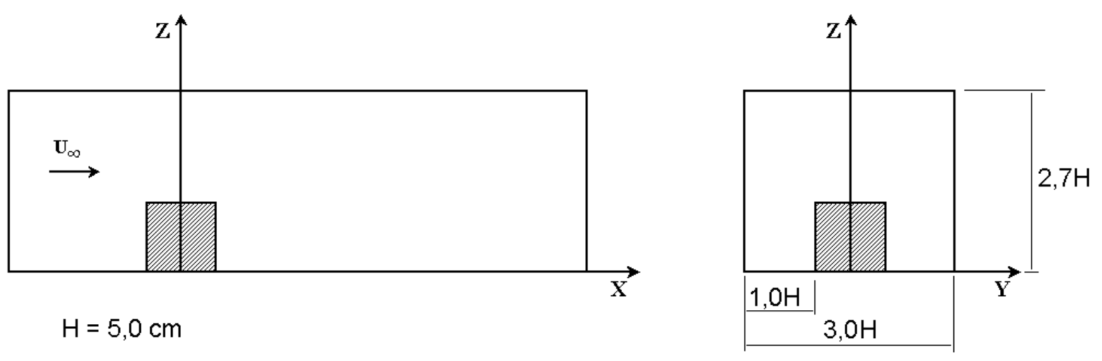

Figura 1 – Descrição geométrica do sistema de coordenadas no túnel de vento

Fonte: Fontoura (2022)

1.1 Extração e organização de dados

Cada medição realizada gerou um arquivo `.dat` com duas colunas: a primeira referente ao tempo de medição, a segunda referente ao valor de velocidade medida. Para realizar a extração dos dados utilizou-se da função `np.loadtxt(data_path, unpack=True)`, que é uma função originária da biblioteca `numpy` importada como `np`.

A partir dos dados foram criados objetos temporais [1,2] e objetos de análises espaciais/estatísticas (para a análise entre dois objetos temporais). Para criá-los, foi construída uma classe de dados que pode ser vista em 1.1. A classe cria um objeto que armazena as propriedades assim declaradas, para criá-los basta entrar com o caminho (*path*) até o arquivo referente a classe, o nome do dado, um índice, os dados de tempo e os dados referentes as velocidades. Os tipos (*data types*) referentes a cada uma das propriedades foi anotado e `init=0` significa que os respectivos dados não serão necessários para a criação da classe e serão atribuídos posteriormente.

Código 1.1 – Código em Python Classe Temporal

```python
from dataclasses import dataclass, field

import numpy as np


@dataclass
class TemporalData:

    path: str
    name: str
    index: str
    u: np.ndarray
    times: np.ndarray
    final_time: float = field(init=0)
    time_step: float = field(init=0)
    N: int = field(init=0)
    u_bar_t: float = field(init=0)
    u_prime: np.ndarray = field(init=0)
    kinetic_energy: float = field(init=0)
    variance: float = field(init=0)
    u_rms: float = field(init=0)
    turb_int: float = field(init=0)
    diss_coef: float = field(init=0)
```

[1] Objetos em python são similares a dicionários, eles armazenam dados com referência a uma chave, contudo podem também ter métodos(funções) específicas a eles. Um objeto pode ser compreendido da seguinte forma: se um veículo é um classe, um carro é uma instância e um objeto dessa classe, podendo armazenar informações como a cor, o modelo, a potência do motor, etc. e é claro tem seus métodos como acelerar, freiar. Uma moto específica pode ser uma instância de uma classe de veículos, mas como ela pertence a outra subclasse ela não possui métodos como ligar o ar condicionado.

[2] As propriedades de um objeto podem ser acessadas por: `propriedade = objeto.propriedade`. Para o objeto exemplificado `potencia = ford_ka.potencia`. Os métodos de um objeto podem ser acessados por: `ford_ka.ligar_ac(True)`

```
    flat_coef: float = field(init=0)
    u_x_pdf : np.ndarray = field(init=0)
    u_pdf: np.ndarray = field(init=0)
    u_prime_x_pdf : np.ndarray = field(init=0)
    u_prime_pdf: np.ndarray = field(init=0)
```

Em seguida foi criada outra classe `TemporalDatas` que é um conjunto de dados, ou seja, serve para criar objetos que armazenam propriedades referentes a análises entre dados temporais. A criação de tais classes pode ser verificada em 1.1. Para criar os objetos são requeridos dois argumentos, são estes o nome da análise e uma lista com os objetos temporais a serem analisados `data_arr`.

Código 1.2 – Código em Python Classe de Análises Espaciais e Estatísticas

```

from dataclasses import dataclass, field

import numpy as np


@dataclass
class TemporalDatas:
    name : str
    data_arr : list
    path : str = field(init=0)
    times : np.ndarray = field(init=0)
    N : int = field(init=0)
    N_u : int = field(init=0)
    final_time : float = field(init=0)
    time_step : np.ndarray = field(init=0)
    u : np.ndarray = field(init=0)
    u_prime : np.ndarray = field(init=0)
    kinetic_energy: np.ndarray = field(init=0)
    variance: np.ndarray = field(init=0)
    u_rms: np.ndarray = field(init=0)
    turb_int: np.ndarray = field(init=0)
    diss_coef: float = field(init=0)
    flat_coef: float = field(init=0)
    u_bar_s : np.ndarray = field(init=0)
    u_prime_bar_s : np.ndarray = field(init=0)
    u_bar_t : float = field(init=0)
    cov : float = field(init=0)
    corr_coef: float = field(init=0)
    positions: list = field(init=0)
```

Para pegar os caminhos (*paths*) que devem ser passados para a função `np.loadtxt(path)` foi utilizado um algorítmo tal que para cada pasta selecionada na `main.py`, os arquivos e pastas dentro dela sejam listados, comparados para que só os arquivos sejam selecionados e em seguida armazenados em uma lista com todos os *paths* referentes a todos os arquivos da referida pasta. Os nomes dos arquivos serão os nomes dos objetos e o índice foi retirado do próprio nome dos arquivos.

Os códigos 1.1 e 1.1 são versões truncadas das declarações de classes implementadas, já que elas possuem métodos extensos para:

- o cálculo das propriedades listadas nos items 1-10 do roteiro;
- a exportação dos dados referentes as propriedades em `.txt` para facilitar a consulta;
- a exportação para `.json` (JavaScript Object Notation), que facilita a manipulação posterior dos dados em um código;
- a conversão para `pd.DataFrame`, que é um formato da biblioteca `pandas` que permite a exportação para formatos como `.csv` e `.xlsx` (Excel).

O código fonte para a geração de classes pode ser consultado no apêndice A.1.

1.2 Função Densidade de Probabilidade

A fim de caracterizar uma variável aleatória, utiliza-se de métodos probabilísticos, uma vez que é impossível determinar um valor para tal variável, uma vez que aleatoriedade significa que a variável não assume nenhum valor certo e nenhum impossível.

A função de densidade de probabilidade, que pode ser compreendida como a probabilidade de se obter um determinado valor para uma determinada variável dentre um determinado espaço amostral para um determinado número de amostragens. Para calcular a probabilidade de um evento ocorrer, pode-mos dizer que é o número de vezes que tal evento ocorre dividido pelo número total de eventos que ocorreram, então a probabilidade de uma velocidade U ser menor que um valor de referência V, isto é, $U \in (-\infty, V)$ pode ser escrita como:

$$p = PU < V$$

A função cumulativa de distribuição de probabilidade é definida como:

$$F(V) \equiv PU < V$$

entre dois valores de referência:

$$PV_a \leq U < V_b = F(V_b) - F(V_a) = \int_{V_a}^{V_b} f(V)dV$$

logo, a função densidade de probabilidade é definida como:

$$f(V) \equiv \frac{dF(V)}{dV} \tag{1.1}$$

Para o caso analisado neste trabalho, soma-se os intervalos de tempo Δt em que a velocidade U assume valores em um ΔU e divide-se pelo tempo total analisado T. Como, o Δt é constante, a densidade de probabilidade é equivalente a um histograma normalizado, já que o histograma é a contagem de valores que caem em um determinado intervalo, normalizando ele é possível obter a função de densidade de probabilidade discreta. A função `pyplot.hist(x, N, density=True)` da biblioteca `matplotlib` serve para tal função.

1.2.1 Função de Densidae de Probabilidade Gaussiana

Variáveis perfeitamente aleatórias podem ser representadas pela função densidade de probabilidade gaussiana, que pode ser definida como:

$$f_{\text{gauss}}(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\left[\frac{(x-\mu)^2}{2\sigma^2}\right]} \tag{1.2}$$

onde μ é a média dos valores assumidos por x e σ é o desvio padrão dos valores assumidos por x, seja x uma variável perfeitamente aleatória.

Desta forma a gaussiana pode ser implementada por meio de:

Código 1.3 – Implementações da Função Densidade de Probabilidade Gaussiana

```
def gaussian(x, mu, sigma):
    """
    Gaussian function
    """
    return
        (1/(sigma*np.sqrt(2*np.pi)))*np.exp(-((x-mu)**2)/(2*sigma**2))

gauss = norm.pdf(x, mu, sigma)
```

onde `norm.pdf(x, mu, sigma)` é uma função da biblioteca `scipy` para o cálculo da função gaussiana. A última implementação foi empregada para a plotagem dos gráficos.

1.3 Médias

Para caracterizar variáves aleatória, tais como valores de velocidade em regime de escoamento turbulento, são frequentemente utilizadas abordagens estatísticas para decompor a velocidade e analisar o escoamento. A decomposição da velocidade u pode ser feita segundo (VERSTEEG; MALALASEKERA, 1995) da seguinte forma:

$$U = \langle U \rangle + u' \tag{1.3}$$

em que $\langle u \rangle$ denota a média do valores assumidos por u e u' denota a flutuação desses valores com relação à média. Disto surge a questão de como calcular a média.

1.3.1 Média Temporal

Segundo (VERSTEEG; MALALASEKERA, 1995), podemos calcular uma média temporal de uma propriedade qualquer $\phi(x,t)$ como segue:

$$\langle \phi(x,t_i) \rangle = \frac{1}{\Delta t} \int_0^{\Delta t} \phi(x,t)dt \tag{1.4}$$

discretizando para o vetor de dados disponíveis, essa equação se torna:

$$\langle \phi \rangle = \frac{1}{T} \sum_{i=1}^{N} \phi_i \Delta t_i \tag{1.5}$$

já que o intervalo de tempo em 1.4, Δt no caso se refere ao tempo de aferição de velocidade, isto é, T, e N se refere ao número de aferições realizadas neste intervalo de tempo $[0, T$. Como o intervalo entre medições Δt é constante e,

$$N = \frac{T}{\Delta t}$$

não é difícil verificar que:

$$\langle \phi \rangle = \frac{1}{N} \sum_{i=1}^{N} \phi_i \tag{1.6}$$

isso significa, para um número de medições N_s somar os valores medidos para a velocidade para cada valor de tempo e dividir por N_s. O que nos mostrará a tendência da velocidade para cada tempo medido.

A implementação em python pode ser feita por meio da seguinte forma:

Código 1.4 – Código em Python Média Temporal

```python
import numpy as np

def temporal_mean(phi:np.ndarray):

    N = len(phi)
    phi_bar = 0

    for i in range(N):
        phi_bar +=  phi[i-1]

    phi_bar /= i

    return phi_bar
```

1.3.2 Média Estatística

De forma análoga, pode-se encontrar a média estatística através da média aritmética das propriedades em um dado ponto no espaço x_i em um dado ponto no tempo t_i:

$$\langle \phi(x_i, t_i) \rangle = \frac{1}{N_s} \sum_{i=1}^{N_s} \phi(x_i, t_i) \tag{1.7}$$

A implementação em python pode ser feita por meio da seguinte forma:

Código 1.5 – Código em Python Média Estatística

```python
import numpy as np

def statistical_mean(data_list):

    statistical_mean = np.zeros(len(data_list[0].u))

    for j in range(len(statistical_mean)):
        spacial_avg = 0.
        for i, data in enumerate(data_list):
            spacial_avg += data.u[j]
            statistical_mean[j] = spacial_avg / len(data_list)

    return statistical_mean
```

Neste código, como visto anteriormente, `data_list` é uma lista com objetos referentes aos dados a serem analisados, isto é cada dado na lista possui propriedades como uma *array* de velocidades `u`, podendo então ser acessada por `data_list[0].u` , para o primeiro elemento da lista. O número de dados temporais analisados é portanto `len(data_list`, o tamanho da minha lista de dados analisados.

1.3.3 Média Espacial

De forma análoga, pode-se encontrar a média espacial através da média aritmética das propriedades em um conjunto de ponto no espaço x em um dado ponto no tempo t_i:

$$\langle \phi(x, t_i) \rangle = \frac{1}{N_p} \sum_{i=1}^{N_p} \phi(x, t_i) \tag{1.8}$$

em que N_p denota o número de pontos no espaço analisados.

De forma similar a média estatística, estamos calculando a média aritmética de valores de velocidade em listas de dados diferentes, então da forma que o código final foi arquitetado, não há a necessidade de implementar uma função diferente para o cálculo da média espacial, pois em essência algoritimicamente, a mesma análise seria realizada. Logo, a função `statistical_mean()` serve para o cálculo da média entre os valores de propriedades de dois ou mais objetos de dados.

1.4 Flutuações

As flutuações ou momentos centrais de primeira ordem podem ser definidos como o quanto um valor particular de U varia de sua média temporal $\langle U \rangle$, após uma manipulação algébrica chega-se em:

$$u_i' = U_i - \langle U \rangle \tag{1.9}$$

Código 1.6 – Código em Python Cálculo de Flutuações

```
import numpy as np
def calculate_fluctuation(u:np.ndarray, u_bar:float):

    return u - u_bar
```

1.5 Energia Cinética de Turbulência por Unidade de Massa

A energia cinética de turbulência (POPE et al., 2000) é calculada por:

$$k = \frac{1}{2}\langle u_i' u_i' \rangle$$

para a análise unidimensional realizada:

$$k = \frac{1}{2}\langle (u')^2 \rangle \tag{1.10}$$

Código 1.7 – Função Cálculo de Intensidade de Turbulência

```
def calculate_kinetic_energy(u:np.ndarray):

    kinetic = 0.5 * calculate_ordered_moment(u, 2)

    return kinetic
```

1.6 Intensidade de Turbulência

A intensidade de turbulência pode ser calculada por:

$$I_i = \frac{\sigma_{u,i}}{\langle U_i \rangle} \tag{1.11}$$

que descreve a relação entre o desvio padrão σ_u ae a média de velocidades em um mesmo eixo.

Código 1.8 – Função Cálculo de Intensidade de Turbulência

```
def calculate_turbulence_intensity(u_bar:float,
   u_prime:np.ndarray):

    u_rms = calculate_std_dev(u_prime)

    return u_rms / np.abs(u_bar)
```

1.7 Momentos de Ordem Superior

1.7.1 Variância

Segundo (POPE et al., 2000) podemos calcular a variância da seguinte forma:

$$\mathrm{var}(U) \equiv \langle (u')^2 \rangle = \int_{-\infty}^{\infty} (V - \langle U \rangle)^2 f(V) dV \tag{1.12}$$

onde V é um intervalo de amostragem qualquer para a velocidade e $f(V)$ é a função densidade de probabilidade para esse intervalo.O que equivale a dizer para uma função discreta que:

$$\langle (u')^2 \rangle = \frac{1}{N} \sum_{i=1}^{N} (u_i')^2 \tag{1.13}$$

Código 1.9 – Cálculo da Variância e Momentos de Ordem Superior

```
import numpy as np
from statistical_functions import temporal_mean

def calculate_ordered_moment(phi_prime:np.ndarray, order: int):

    variance = temporal_mean(phi_prime ** order)

    return variance
```

1.7.2 Desvio Padrão

O desvio padrão é definido como a raiz quadrada da variância, logo:

$$\sigma_u = \sqrt{\langle (u')^2 \rangle} \tag{1.14}$$

Código 1.10 – Função Cálculo de Desvio Padrão

```
def calculate_std_dev(phi_prime:np.ndarray):
```

```python

    variance = calculate_ordered_moment(phi_prime, 2)

    return np.sqrt(variance)
```

1.7.3 Coeficiente de Assimetria (*Skewness*)

Frequentemente é conveniente utilizar variáveis normalizadas, isto é, variáveis que tem média 0 ($\langle\hat{\phi}\rangle = 0$) e variância unitária ($\hat{\sigma_\phi} = 1$). Para isto fazemos:

$$\hat{\phi} = \frac{(\phi - \langle\phi\rangle)}{\sigma_\phi}$$

Segundo (POPE et al., 2000) para casos gerais os momentos de ordem superior normalizados podem ser calculados por:

$$\hat{\mu}_n = \frac{\langle(u')^n\rangle}{\sigma_u^n}$$

Como a média de variáveis aleatórias centradas não contém informações relevantes para quaisquer análises, além da variância são empregados o coeficiente de assimetria (*skewness*) e o coeficiente de achatamento (*kurtosis*) para tal fim. O coeficiente de assimetria é um momento de terceira ordem e pode portanto ser calculado por:

$$S = \hat{\mu}_3 = \frac{\langle(u')^3\rangle}{(\sigma_u)^3} \tag{1.15}$$

Código 1.11 – Código em Python Cálculo de Coeficiente de Assimetria

```python
def calculate_dissimetry_coef(phi_prime:np.ndarray):

    sigma_3 = calculate_ordered_moment(phi_prime, 3)
    sigma_2 = calculate_ordered_moment(phi_prime, 2)

    return sigma_3 / ((sigma_2)**(3/2))
```

1.7.4 Coeficiente de Achatamento (*Kurtosis*)

O coeficiente de achatamento é um momento de quarta ordem e pode portanto ser calculado por:

$$T = \hat{\mu}_4 = \frac{\langle(u')^4\rangle}{(\sigma_u)^4} \tag{1.16}$$

Código 1.12 – Código em Python Cálculo de Achatamento

```python
def calculate_flatenning_coef(phi_prime:np.ndarray):
```

```

    sigma_4 = calculate_ordered_moment(phi_prime, 4)
    sigma_2 = calculate_ordered_moment(phi_prime, 2)

    return sigma_4 / (sigma_2**2)
```

1.8 Covariância e Correlação

1.8.1 Covariânia

A covariância é um momento de segunda ordem misto que mostra como duas variáveis aleatórias centradas se relacionam entre si. Segundo (POPE et al., 2000), ela pode ser calculada por:

$$\mathrm{cov}(U_1, U_2) = \langle u_1' u_2' \rangle = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} (V_1 - \langle U_1 \rangle)(V_2 - \langle U_2 \rangle) f_{12}(V_1, V_2) dV_1 dV_2 \tag{1.17}$$

onde $f_{12}(x_1,x_2)$ é a função densidade de probabilidade para duas variáveis. É importante ressaltar que $f_{12}(x_1, x_2) = f_1(x_1) f_2(x_2)$, além disso:

$$f_1(x_1) = \int_{-\infty}^{\infty} f_{12}(x_1, x_2) dx_1$$

e para uma função qualquer $Q(U)$:

$$\langle Q(U) \rangle = \int_{-\infty}^{\infty} Q(V) f(V) dV$$

Note que $\mathrm{cov}(U_1, U_1) \equiv \mathrm{var}(U_1)$.

Disto é possível calcular algoritimicamente a covariância como segue:

Código 1.13 – Função Cálculo da Covariância

```
def covariance(a_prime:np.ndarray, b_prime:np.ndarray):
    if len(a_prime) != len(b_prime):
        raise ValueError('Matrices must have the same length')

    cov = 0
    for a_p, b_p in zip(a_prime, b_prime):
        cov += a_p * b_p
    cov /= len(a_prime)

    return cov
```

onde a função `zip()` é utilizada para pegar os elementos das matrizes `a_prime` e `b_prime` que serão utilizados em cada iteração. A primeira condicional checa se os tamanhos das matrizes são iguais e gera um erro se não forem.

1.8.2 Coeficiente de Correlação

Segundo (POPE et al., 2000) e (FONTOURA, 2022) o coeficiente de correlação de duas variáveis pode ser calculado por:

$$R_{12} = \frac{\langle u_1' u_2' \rangle}{\sigma_{u,1}\sigma_{u,2}} \tag{1.18}$$

Em geral, tem-se uma desigualdade de Cauchy-Schwartz, isto é $R_{12} \in [-1,1]$. Pode-se calcular numéricamente o coeficiente de correlação então da seguinte forma:

Código 1.14 – Função Cálculo do Coeficiente de Correlação

```python
def correlation_coeff(a_prime:np.ndarray, b_prime:np.ndarray):

    cov_ab = covariance(a_prime,b_prime)
    a_rms = calculate_std_dev(a_prime)
    b_rms = calculate_std_dev(b_prime)
    corr_coef = cov_ab / (a_rms * b_rms)

    return corr_coef
```

2 Metodologia Numérica e Resultados

2.1 Metodologia Numérica

Para o cálculo das propriedades de cada arquivo de projeto, foi definido um método na classe `TemporalData` como segue:

Código 2.1 – Método para Cálculo de Atributos da Classe para Dados Temporais

```python
def __post_init__(self) -> None:
        self.N = len(self.u)
        self.final_time = self.times[-1]
        self.time_step = round(self.times[-1] - self.times[-2], 4)
        self.calculate_properties()

        if self.name[:-1] == 'lre' or self.name[:-1] == 'hre':
            self.index = self.index

    def calculate_properties(self) -> None:

        self.u_bar_t = temporal_mean(self.u)
        self.u_prime = calculate_fluctuation(self.u, self.u_bar_t)
        self.kinetic_energy = .5 * calculate_ordered_moment(self.u_prime, 2)
        self.variance = calculate_ordered_moment(self.u_prime, 2)
        self.u_rms = np.sqrt(calculate_ordered_moment(self.u_prime, 2))
        self.turb_int = calculate_turbulence_intensity(self.u_bar_t, self.u_prime)
        self.diss_coef = calculate_dissimetry_coef(self.u_prime)
        self.flat_coef = calculate_flatenning_coef(self.u_prime)
        self.u_x_pdf, self.u_pdf = pdf(self.u, 500)
        self.u_prime_x_pdf, self.u_prime_pdf = pdf(self.u_prime)
```

Para o cálculo das propriedades de cada análise entre dados temporais foi definido o seguinte método para a classe `TemporalDatas`:

Código 2.2 – Método para Cálculo de Atributos da Classe para Dados Estatísticos/Espaciais

```python
    def __post_init__(self):

        self.data_arr.sort(key= lambda x: x.path)
        self.path = "/".join(self.data_arr[0].path.split('/')[:-1])+f'/{self.name}'
        self.times = self.data_arr[0].times
```

```
        self.N = len(self.data_arr)
        self.N_u = len(self.data_arr[0].u)
        self.final_time = self.times[-1]
        self.time_step = round(self.times[-1] - self.times[-2], 4)
        self.u = np.array([data.u for data in self.data_arr])
        self.u_bar_s = statistical_mean(self.data_arr)
        self.u_bar_t = temporal_mean(self.u_bar_s)
        self.u_prime = calculate_fluctuation(self.u, self.u_bar_s)
        self.u_prime_bar_s = calculate_fluctuation(self.u_bar_s,
           self.u_bar_t)
        self.cov = 0.
        self.corr_coef = 0.
        self.positions = [0.]
        self.calculate_properties()

    def calculate_properties(self) -> None:
        self.variance = calculate_ordered_moment(self.u_prime, 2)
        self.u_rms =
           np.sqrt(calculate_ordered_moment(self.u_prime_bar_s, 2))
        self.kinetic_energy =
           calculate_kinetic_energy(self.u_prime)
        self.turb_int =
           calculate_turbulence_intensity(self.u_bar_s,
           self.u_prime_bar_s)
        self.diss_coef =
           calculate_dissimetry_coef(self.u_prime_bar_s)
        self.flat_coef =
           calculate_flatenning_coef(self.u_prime_bar_s)
        self.u_x_pdf, self.u_pdf = pdf(self.u_bar_s, 500)
        self.u_prime_x_pdf, self.u_prime_pdf =
           pdf(self.u_prime_bar_s)

    def calculate_cov_corr(self, index_a, index_b):

        self.cov = self.covariance(index_a, index_b)
        self.corr_coef = self.correlation(index_a, index_b)

        return (self.cov, self.corr_coef)

    def covariance(self, index_a:int, index_b:int):

        cov = covariance(self.data_arr[index_a].u_prime,
                                     self.data_arr[index_b].u_prime)

        return cov

    def correlation(self, index_a:int, index_b:int):
        cov = covariance(self.data_arr[index_a].u_prime,
                         self.data_arr[index_b].u_prime)

        corr_coef = cov / (self.data_arr[index_a].u_rms *
           self.data_arr[index_b].u_rms)
```

```
        return corr_coef
```

É importante ressaltar que as funções para o cálculo de propriedades não aparecem explicitamente, pois foram importadas de outro arquivc `statistical_functions.py` que pode ser encontrado no apêndice A.2. Isto foi feito para fins de organização e para evitar erros provenientes de mudanças acidentais nos algorítmos.

2.2 Resultados

2.2.1 Dados lre

Foram encontrados para os dados referêntes ao baixo número de Reynolds os valores descritos na tabela 1. Os gráficos gerados para a velocidade instantânea pode ser observada nas figuras 2-6. As covariâncias e correlações espaciais estão descritos na tabela 2. O número de Reynolds para o escoamento lre é dado por:

$$Re = \frac{lu}{\nu} \tag{2.1}$$

em que l é a escala de comprimento proporcional a altura do canal plano, u é a escala de velocidade proporcional a velocidade média do escoamento e ν é a viscosidade cinemática $\nu = \rho/\mu$.

Propriedades	u_1	u_2	u_3	u_4	u_5
Velocidade Média Temporal	1.713823	0.950646	0.983968	1.039745	1.051171
Variância	0.000194	0.025693	0.019978	0.024767	0.031191
Desvio$_{p}adr\cap o$	0.013940	0.160290	0.141345	0.157375	0.176610
Intensidade de Turbulência	0.008134	0.168611	0.143648	0.151359	0.168012
Energia Cinética de Turbulência	0.000097	0.012846	0.009989	0.012383	0.015595
Coeficiente de Dissimetria	-0.041953	-0.424089	-0.288782	-0.259762	0.793274
Coeficiente de Achatamento	3.406720	3.531700	3.721351	4.947793	7.134215

Tabela 1 – Tabela de Resultados Para lre

Propriedades	$u_1'u_2'$	$u_2'u_3'$	$u_3'u_4'$	$u_4'u_5'$
Covariâncias Espaciais	0.000376	-0.000251	-7.818e-05	0.002757
Correlações Espaciais	0.168236	-0.011096	-0.003515	0.099202

Tabela 2 – Tabela de Covariâncias e Correlações Para lre

2.2.2 Dados hre

Foram encontrados para os dados estatísticos referêntes ao alto número de Reynolds os valores descritos na tabela 3. Os gráficos gerados para a velocidade instantânea pode ser observada nas figuras 12-16. As covariâncias e correlações espaciais estão descritos na tabela 4. Os gráficos comparando a distribuição real de velocidades com as respectivas curvas gaussianas podem ser encontradas nas figuras 17 - 21.

Propriedades	u_1	u_2	u_3	u_4	u_5
Velocidade Média Temporal	11.644965	5.094543	6.150405	6.364920	6.247362
Variância Temporal	0.082662	5.383083	6.360148	6.318512	6.407064
Desvio Padrão	0.287510	2.320147	2.521933	2.513665	2.531218
Intensidade de Turbulência	0.024690	0.455418	0.410043	0.394925	0.405166
Energia Cinética de Turbulência	0.041331	2.691541	3.180074	3.159256	3.203532
Coeficiente de Dissimetria	-0.367314	0.402022	0.304111	0.272568	0.229162
Coeficiente de Achatamento	1.659989	2.690877	2.487638	2.497051	2.458957

Tabela 3 – Tabela de Resultados Para hre

Propriedades	$u'_1u'_2$	$u'_2u'_3$	$u'_3u'_4$	$u'_4u'_5$
Covariâncias Espaciais	-0.000369	-0.114053	0.0328592	-0.100446
Correlações Espaciais	-0.000553	-0.019492	0.005183	-0.015786

Tabela 4 – Tabela de Covariâncias e Correlações Para hre

2.2.3 Resultados hre-prob

Foi encontrado para os dados referêntes ao alto número de Reynolds os valores descritos na tabela 3. Os gráficos gerados para a velocidade instantânea pode ser observada nas figuras 12-16. As covariâncias e correlações espaciais estão descritos na tabela 4. Os gráficos comparando a distribuição real de velocidades com as respectivas curvas gaussianas podem ser encontradas nas figuras 17 - 21

Propriedades	Valor Estatístico
Velocidade Média	6.254618
Variância	6.597853
Desvio Padrão	2.568509
Intensidade de Turbulência	0.410708
Energia Cinética de Turbulência	3.298926
Coeficiente de Dissimetria	0.262753
Coeficiente de Achatamento	2.428180

Tabela 5 – Tabela de Resultados Para hre-prob

2.2.4 Perfil Montante

Conjunto de dados contidos nas pastas `perfil_mon` e `perfil_jus` são destinados à plotagem dos perfis de velocidade média e de intensidade de turbulência. Neste sentido foram traçado os quatro perfis sendo eles as figuras 24-27.

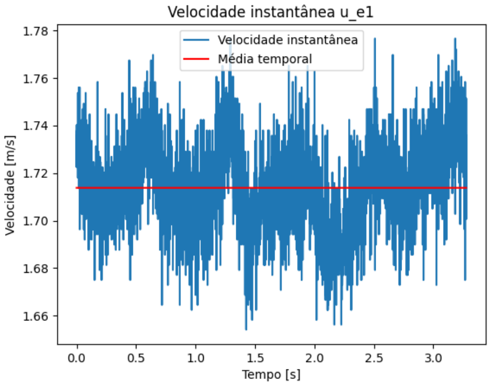

Figura 2 – Gráfico Velocidades Instantâneas lre1

Fonte: Produzido pelo autor

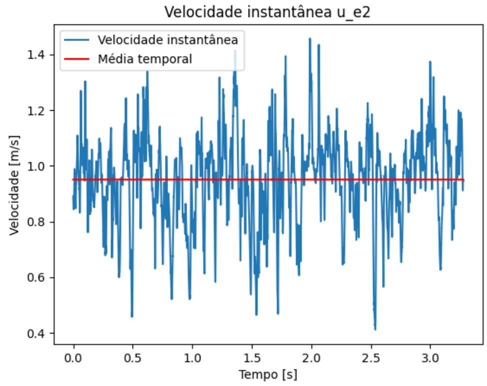

Figura 3 – Gráfico Velocidades Instantâneas lre2

Fonte: Produzido pelo autor

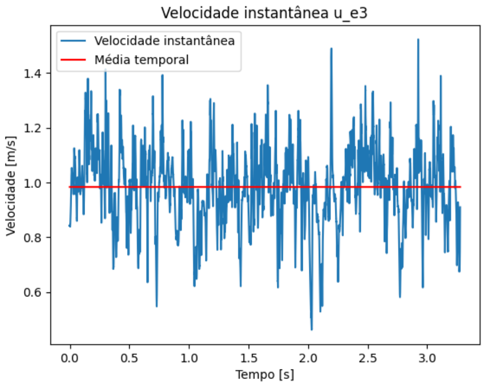

Figura 4 – Gráfico Velocidades Instantâneas lre3

Fonte: Produzido pelos autores

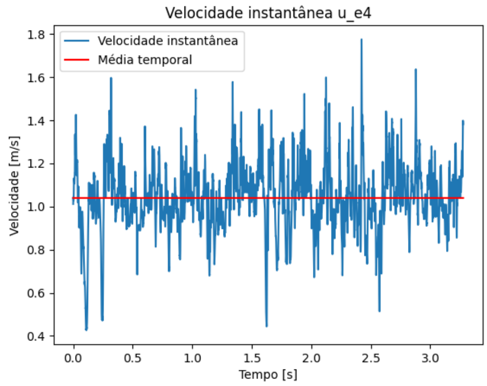

Figura 5 – Gráfico Velocidades Instantâneas lre4

Fonte: Produzido pelo autor

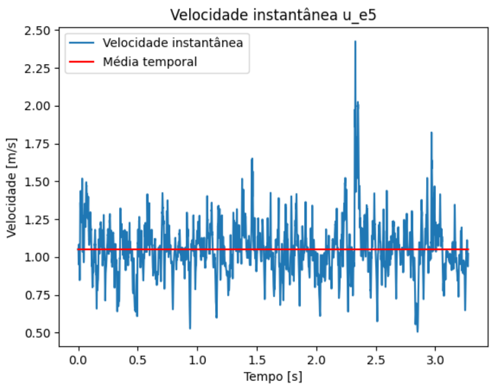

Figura 6 – Gráfico Velocidades Instantâneas lre5

Fonte: Produzido pelos autores

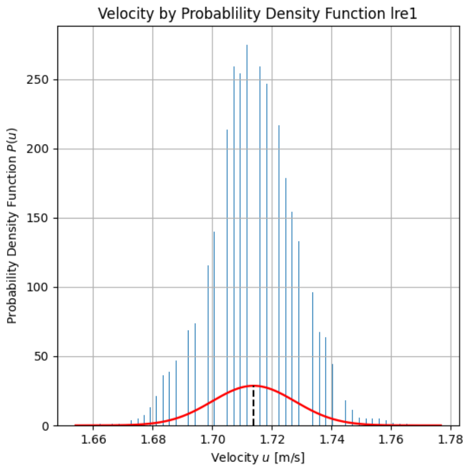

Figura 7 – Gráfico Densidade de Probabilidade lre1

Fonte: Produzido pelo autor

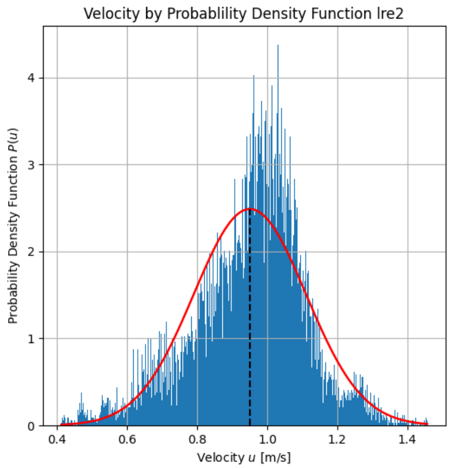

Figura 8 – Gráfico Densidade de Probabilidade lre2

Fonte: Produzido pelo autor

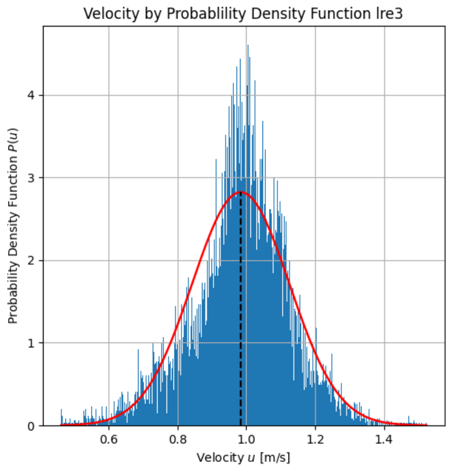

Figura 9 – Gráfico Densidade de Probabilidade lre3

Fonte: Produzido pelo autor

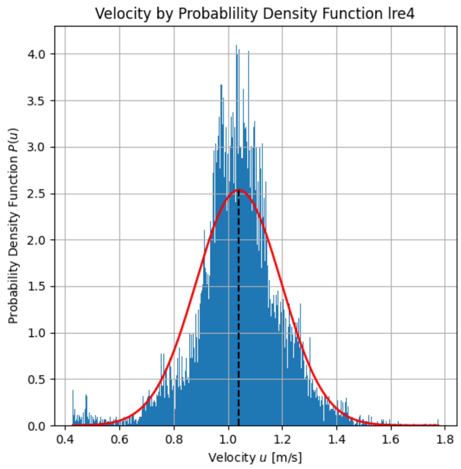

Figura 10 – Gráfico Densidade de Probabilidade lre4

Fonte: Produzido pelo autor

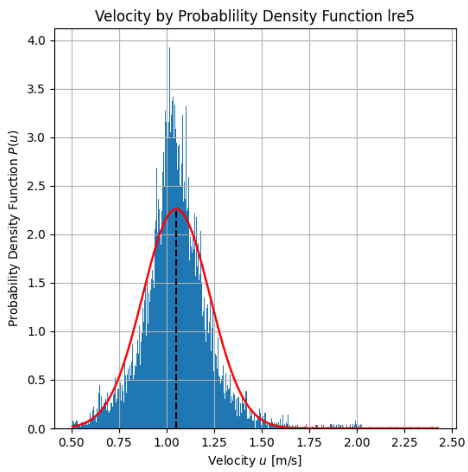

Figura 11 – Gráfico Densidade de Probabilidade lre4

Fonte: Produzido pelo autor

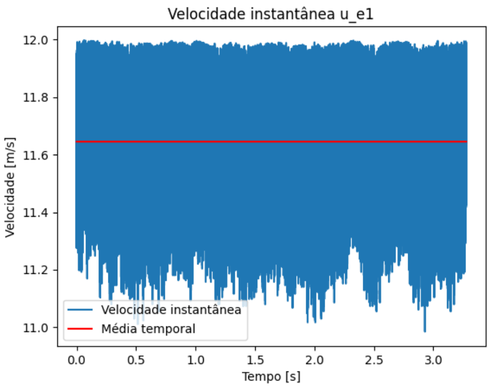

Figura 12 – Gráfico Velocidades Instantâneas hre1

Fonte: Produzido pelo autor

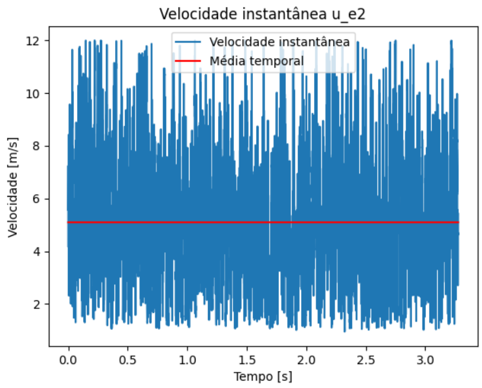

Figura 13 – Gráfico Velocidades Instantâneas hre2

Fonte: Produzido pelo autor

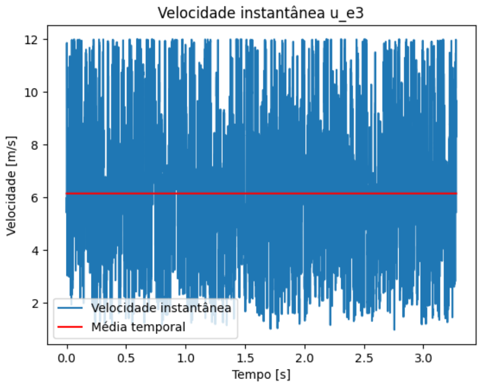

Figura 14 – Gráfico Velocidades Instantâneas hre3

Fonte: Produzido pelos autores

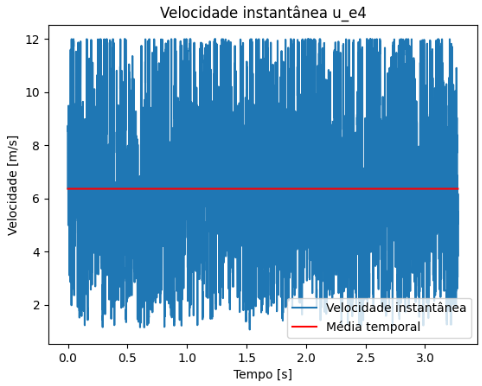

Figura 15 – Gráfico Velocidades Instantâneas hre4

Fonte: Produzido pelo autor

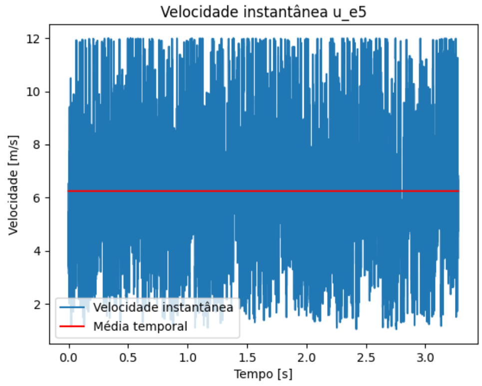

Figura 16 – Gráfico Velocidades Instantâneas hre5

Fonte: Produzido pelos autores

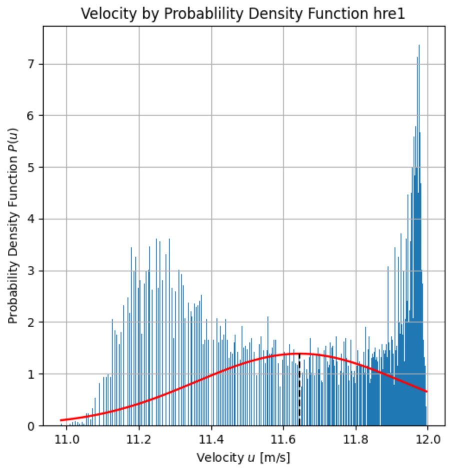

Figura 17 – Gráfico Densidade de Probabilidade hre1

Fonte: Produzido pelo autor

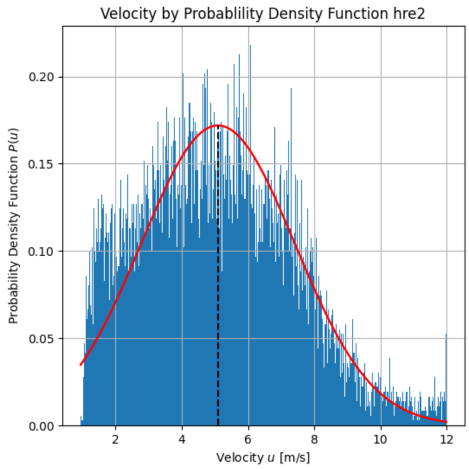

Figura 18 – Gráfico Densidade de Probabilidade hre2

Fonte: Produzido pelo autor

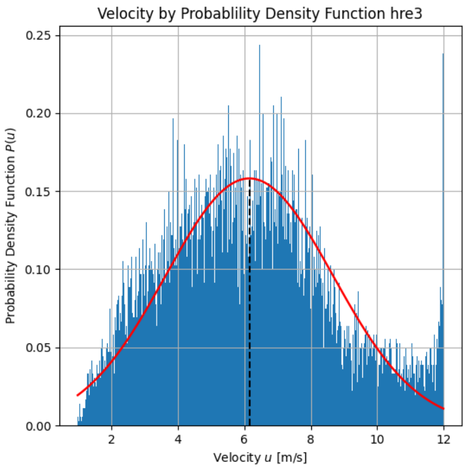

Figura 19 – Gráfico Densidade de Probabilidade hre3

Fonte: Produzido pelo autor

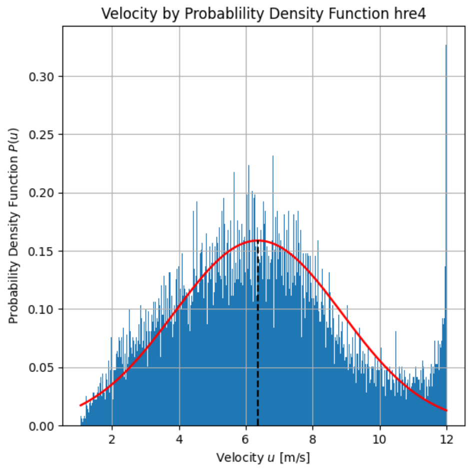

Figura 20 – Gráfico Densidade de Probabilidade hre4

Fonte: Produzido pelo autor

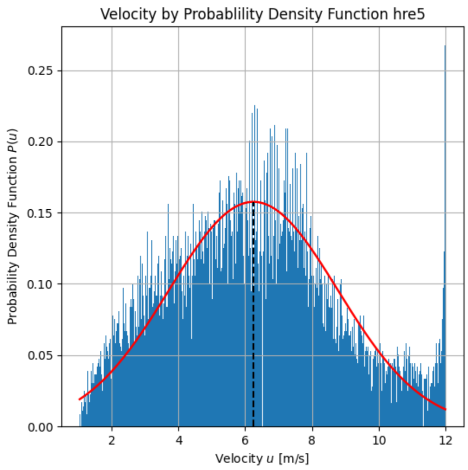

Figura 21 – Gráfico Densidade de Probabilidade hre4

Fonte: Produzido pelo autor

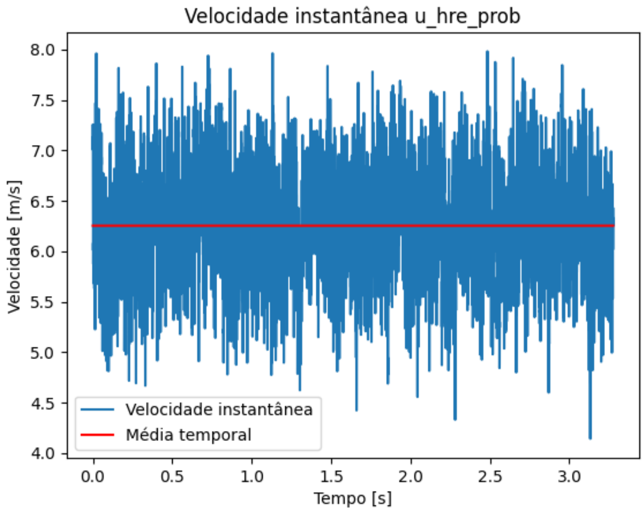

Figura 22 – Gráfico Velocidades Médias hre-prob

Fonte: Produzido pelo autor

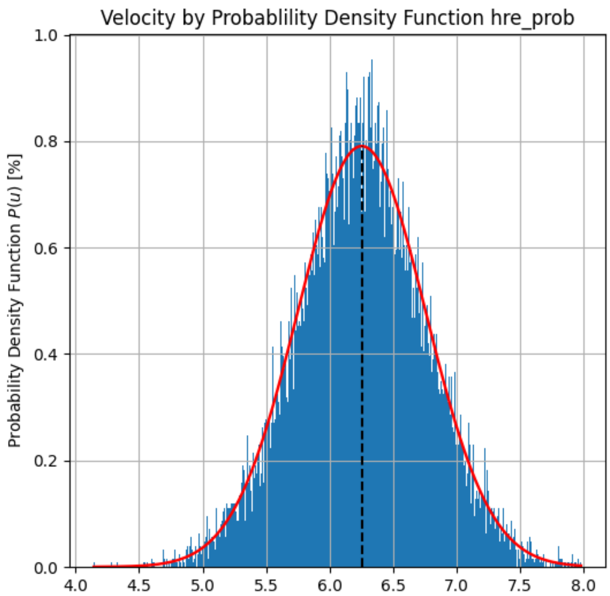

Figura 23 – Gráfico Densidade de Probabilidade hre-prob

Fonte: Produzido pelo autor

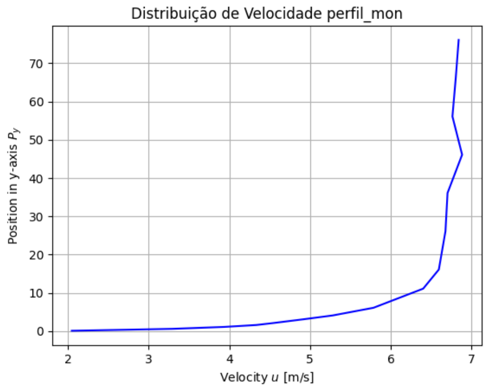

Figura 24 – Gráfico Perfil de Velocidade Média

Fonte: Produzido pelo autor

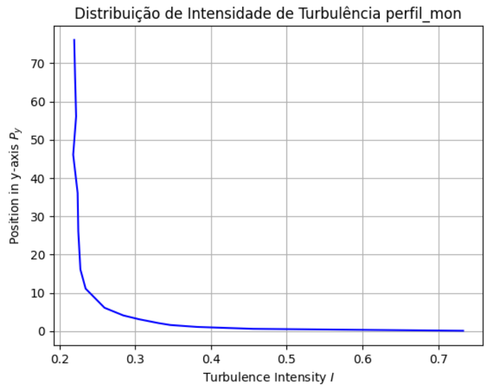

Figura 25 – Gráfico Perfil de Intensidade de Turbulência

Fonte: Produzido pelo autor

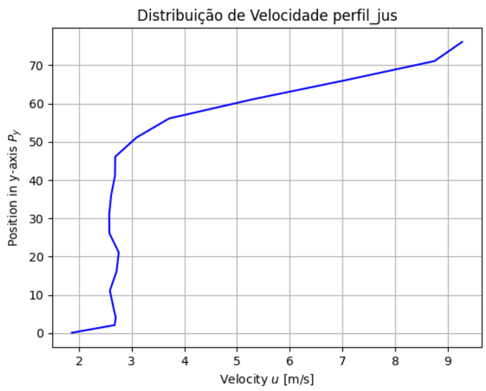

Figura 26 – Gráfico Perfil de Velocidade Média

Fonte: Produzido pelo autor

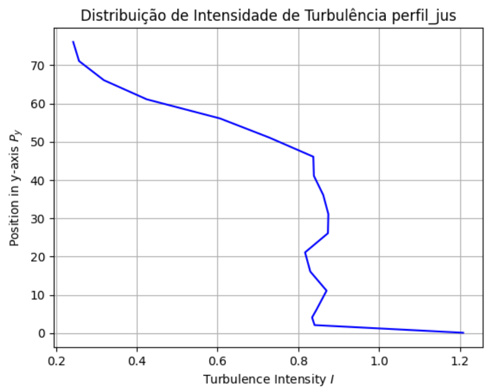

Figura 27 – Gráfico Perfil de Intensidade de Turbulência

Fonte: Produzido pelo autor

3 Conclusão

A partir das análises conduzidas ao longo do presente trabalho foi possível observar a influência do corpo robundo na formação de turbulência para o menor número de Reynolds, a influência do número de Reynolds nas propriedades turbulentas do escoamento, o caráter randômico da velocidade para escoamentos turbulentos, uma vez que o gráfico de distribuição de probabilidade se aproxima muito de uma distribuição gaussiana em todos os escoamentos. Além de ver que há uma certa ordem na turbulência como pode ser observado nos coeficientes de correlação e nas covariâncias. Além disso nas análises de `hre_prob` foi possível ver claramente o poder das abordagens estatísticas na caracterização do escoamento e como a média estatística mesmo sendo incrívelmente mais custosa consegue descrever bem as tendências do escoamento. Por fim foi possível ver como o perfil de velocidade média são distindos entre a montante e a jusante do corpo robundo, assim como observar como a velocidade média temporal varia ao longo do eixo z do escoamento. É interessante ver também que a intensidade de turbulência varia ao longo do eixo z e como no geral as características do escoamento turbulento variam ao longo dos eixos e ao longo do tempo.

No geral, acredito que o trabalho agregou incrívelmento no entendimento estatístico com relação a turbulência do autor, isto é o que de fato significa cada propriedade, como elas são calculadas e como elas podem servir de ferramentas para caracterizar modelos numérios e experimentais de escoamentos turbulentos.

3.0.1 Pósfácio

Neste trabalho todas as propriedades do escoamento pedidas forma calculadas através do programa em Python para todos os pontos de dados, bem como algumas análises entre pontos de dados como foi requisitado e extra do que foi requisitado. De qualquer forma o código fonte e sua estrutura de dados pode ser acessada livremente pelo GitHub pelo url: https://github.com/felca25/turb

É claro que como em qualquer trabalho acredito que existem muitos pontos a serem melhorados, como a função densidade de probabilidade, a forma como os dados são calculados, além da fundamentação teórica e organização dos resultados. De qualauer forma considero um trabalho minimamente descente.

Referências

FONTOURA, J. L. A. da. **MEDIÇÃO DE VELOCIDADES TURBULENTAS ANEMOMETRIA DE FIO QUENTE TRATAMENTO ESTATÍSTICO E INTERPRETAÇÃO DOS RESULTADOS OBTIDOS**. 5 jul. 2022. Citado nas pp. 6, 17.

POPE, S.; ECCLES, P.; POPE, S.; PRESS, C. U. **Turbulent Flows**. Cambridge University Press, 2000. ISBN 9780521598866. Disponível em: <https://books.google.de/books?id=HZsTw9SMx-0C>. Citado nas pp. 13–17.

VERSTEEG, H.; MALALASEKERA, W. **An Introduction to Computational Fluid Dynamics: The Finite Volume Approach**. Longman Scientific & Technical, 1995. ISBN 9780582218840. Disponível em: <https://books.google.com.br/books?id=7UynjgEACAAJ>. Citado nas pp. 10, 11.

Apêndices

APÊNDICE A – Código Fonte

A.1 Definição de Classes

Código A.1 – Código em Python para a declaração de classes

```python
from dataclasses import dataclass, field
from matplotlib import pyplot as plt
from scipy.stats import norm
import os, os.path
import numpy as np
import pandas as pd
import json
from statistical_functions import *
from data_man import export_data

@dataclass
class TemporalData:

    path: str
    name: str
    index: str
    u: np.ndarray
    times: np.ndarray
    final_time: float = field(init=0)
    time_step: float = field(init=0)
    N: int = field(init=0)
    u_bar_t: float = field(init=0)
    u_prime: np.ndarray = field(init=0)
    kinetic_energy: float = field(init=0)
    variance: float = field(init=0)
    u_rms: float = field(init=0)
    turb_int: float = field(init=0)
    diss_coef: float = field(init=0)
    flat_coef: float = field(init=0)
    u_x_pdf : np.ndarray = field(init=0)
    u_pdf: np.ndarray = field(init=0)
    u_prime_x_pdf : np.ndarray = field(init=0)
    u_prime_pdf: np.ndarray = field(init=0)

    def __post_init__(self) -> None:
        self.N = len(self.u)
        self.final_time = self.times[-1]
        self.time_step = round(self.times[-1] - self.times[-2], 4)
        self.calculate_properties()

        if self.name[:-1] == 'lre' or self.name[:-1] == 'hre':
            self.index = self.index
```

```python

    def calculate_properties(self) -> None:

        self.u_bar_t = temporal_mean(self.u)
        self.u_prime = calculate_fluctuation(self.u, self.u_bar_t)
        self.kinetic_energy = .5 * calculate_ordered_moment(self.u_prime, 2)
        self.variance = calculate_ordered_moment(self.u_prime, 2)
        self.u_rms = np.sqrt(calculate_ordered_moment(self.u_prime, 2))
        self.turb_int = calculate_turbulence_intensity(self.u_bar_t, self.u_prime)
        self.diss_coef = calculate_dissimetry_coef(self.u_prime)
        self.flat_coef = calculate_flatenning_coef(self.u_prime)
        self.u_x_pdf, self.u_pdf = pdf(self.u, 500)
        self.u_prime_x_pdf, self.u_prime_pdf = pdf(self.u_prime)

    def save_txt(self):
        try:
            os.mkdir('txt_results')
        except FileExistsError:
            pass

        with open(f'txt_results/{self.name}.txt', 'w') as outfile:

            outfile.write(f'{self.name.upper()} results\n\n')
            outfile.write(f'u_bar_t = {self.u_bar_t}\n'\
                          f'variance = {self.variance}\n'\
                            f'std_dev = {self.u_rms}\n'\
                                f'turbulence intensity = {self.turb_int}\n'\
                                    f'dissimetry coefficient = {self.diss_coef}\n'\
                                        f'flatenning coefficient = {self.flat_coef}\n')

        return 1

    def to_list(self)-> None:
        self.times = self.times.tolist()
        self.u = self.u.tolist()
        self.u_prime = self.u_prime.tolist()
        self.u_x_pdf = self.u_x_pdf.tolist()
        self.u_pdf = self.u_pdf.tolist()
        self.u_prime_x_pdf = self.u_prime_x_pdf.tolist()
        self.u_prime_pdf = self.u_prime_pdf.tolist()

    def to_array(self) -> None:
        self.times = np.array(self.times)
        self.u = np.array(self.u)
        self.u_prime = np.array(self.u_prime)
```

```
        self.u_x_pdf = np.array(self.u_x_pdf)
        self.u_pdf = np.array(self.u_pdf)
        self.u_prime_x_pdf = np.array(self.u_prime_x_pdf)
        self.u_prime_pdf = np.array(self.u_prime_pdf)

    def save_json(self):
        self.to_list()
        dir_name = self.path.split('/')[-2]

        try:
            os.mkdir(f'json_results/{dir_name}')
        except FileExistsError:

            pass
        if self.name == 'PERFILM':
            with open(f'json_results/{dir_name}/{self.name}'\
                f'{self.index}.json', 'w') as outfile:
                json.dump(self.__dict__, outfile)
        else:
            with open(f'json_results/{dir_name}/{self.name}.json',
               'w') as outfile:
                json.dump(self.__dict__, outfile)

        self.to_array()


@dataclass
class TemporalDatas:
    name : str
    data_arr : list
    path : str = field(init=0)
    times : np.ndarray = field(init=0)
    N : int = field(init=0)
    N_u : int = field(init=0)
    final_time : float = field(init=0)
    time_step : np.ndarray = field(init=0)
    u : np.ndarray = field(init=0)
    u_prime : np.ndarray = field(init=0)
    kinetic_energy: np.ndarray = field(init=0)
    variance: np.ndarray = field(init=0)
    u_rms: np.ndarray = field(init=0)
    turb_int: np.ndarray = field(init=0)
    diss_coef: float = field(init=0)
    flat_coef: float = field(init=0)
    u_bar_s : np.ndarray = field(init=0)
    u_prime_bar_s : np.ndarray = field(init=0)
    u_bar_t : float = field(init=0)
    cov : float = field(init=0)
    corr_coef: float = field(init=0)
    positions: list = field(init=0)

    def __post_init__(self):
```

```

        self.data_arr.sort(key= lambda x: x.path)
        self.path = "/".join(self.data_arr[0].path.\
            split('/')[:-1])+f'/{self.name}'
        self.times = self.data_arr[0].times
        self.N = len(self.data_arr)
        self.N_u = len(self.data_arr[0].u)
        self.final_time = self.times[-1]
        self.time_step = round(self.times[-1] - self.times[-2], 4)
        self.u = np.array([data.u for data in self.data_arr])
        self.u_bar_s = statistical_mean(self.data_arr)
        self.u_bar_t = temporal_mean(self.u_bar_s)
        self.u_prime = calculate_fluctuation(self.u, self.u_bar_s)
        self.u_prime_bar_s = calculate_fluctuation(self.u_bar_s,
           self.u_bar_t)
        self.cov = 0.
        self.corr_coef = 0.
        self.positions = [0.]
        self.calculate_properties()

    def calculate_properties(self) -> None:
        self.variance = calculate_ordered_moment(self.u_prime, 2)
        self.u_rms =
           np.sqrt(calculate_ordered_moment(self.u_prime_bar_s, 2))
        self.kinetic_energy =
           calculate_kinetic_energy(self.u_prime)
        self.turb_int =
           calculate_turbulence_intensity(self.u_bar_s,
           self.u_prime_bar_s)
        self.diss_coef =
           calculate_dissimetry_coef(self.u_prime_bar_s)
        self.flat_coef =
           calculate_flatenning_coef(self.u_prime_bar_s)
        self.u_x_pdf, self.u_pdf = pdf(self.u_bar_s, 500)
        self.u_prime_x_pdf, self.u_prime_pdf =
           pdf(self.u_prime_bar_s)

    def calculate_cov_corr(self, index_a, index_b):
        print(self.data_arr[index_a].name)
        print(self.data_arr[index_b].name)

        self.cov = self.covariance(index_a, index_b)
        self.corr_coef = self.correlation(index_a, index_b)

        return (self.cov, self.corr_coef)

    def covariance(self, index_a:int, index_b:int):
        print(self.data_arr[index_a].name)
        print(self.data_arr[index_b].name)

        cov = covariance(self.data_arr[index_a].u_prime,
                                self.data_arr[index_b].u_prime)
```

```

        return cov

    def correlation(self, index_a:int, index_b:int):
        cov = covariance(self.data_arr[index_a].u_prime,
                        self.data_arr[index_b].u_prime)

        corr_coef = cov / (self.data_arr[index_a].u_rms * self.data_arr[index_b].u_rms)

        return corr_coef

    def save_txt(self):
        try:
            os.mkdir('txt_results')
        except FileExistsError:
            pass

        with open(f'txt_results/{self.name}.txt', 'w') as outfile:

            outfile.write(f'{self.name.upper()} results\n\n')
            outfile.write(f'u_bar_t = {self.u_bar_t}\n'\
                          f'variance = {self.variance}\n'\
                            f'std_dev = {self.u_rms}\n'\
                                f'turbulence intensity = {self.turb_int}\n'\
                                    f'dissimetry coefficient = {self.diss_coef}\n'\
                                        f'flatenning coefficient = {self.flat_coef}\n')

    def to_list(self):

        for data in self.data_arr:
            data.to_list()

        auxiliary = self.data_arr
        delattr(self, 'data_arr')
        self.times = self.times.tolist()
        self.u = self.u.tolist()
        self.u_bar_s = self.u_bar_s.tolist()
        self.u_prime = self.u_prime.tolist()
        self.u_prime_bar_s = self.u_prime_bar_s.tolist()
        self.kinetic_energy = self.kinetic_energy.tolist()
        self.variance = self.variance.tolist()
        self.turb_int = self.turb_int.tolist()
        self.u_x_pdf = self.u_x_pdf.tolist()
        self.u_pdf = self.u_pdf.tolist()
        self.u_prime_x_pdf = self.u_prime_x_pdf.tolist()
        self.u_prime_pdf = self.u_prime_pdf.tolist()

        return auxiliary
```

```python

    def to_array(self):

        for data in self.data_arr:
                data.to_array()

        self.times = np.array(self.times)
        self.u = np.array(self.u)
        self.u_bar_s = np.array(self.u_bar_s)
        self.u_prime = np.array(self.u_prime)
        self.u_prime_bar_s = np.array(self.u_prime_bar_s)
        self.kinetic_energy = np.array(self.kinetic_energy)
        self.variance = np.array(self.variance)
        self.turb_int = np.array(self.turb_int)
        self.u_x_pdf = np.array(self.u_x_pdf)
        self.u_pdf = np.array(self.u_pdf)
        self.u_prime_x_pdf = np.array(self.u_prime_x_pdf)
        self.u_prime_pdf = np.array(self.u_prime_pdf)

    def save_json(self):

        auxiliary = self.to_list()

        try:
            os.mkdir(f'json_results/{self.name}')
        except FileExistsError:

            pass
        with open(f'json_results/{self.name}/{self.name}.json', 'w') as outfile:
            json.dump(self.__dict__, outfile)


        self.data_arr = auxiliary
        self.to_array()


    def to_data_frame(self, properties):
        name = self.name
        treated = [list(self.data_arr[0].times),]
        headers = ['tempos[s]',]

        if name == 'perfil_mon' or name == 'perfil_jus':

            for property, in zip(properties):
                for i, temporal_data, in enumerate(self.data_arr):
                    if property == 'u':
                        treated.append(list(temporal_data.u))
                    headers.append(f'{property}_{i+1}')

            df = pd.DataFrame(np.transpose(treated), columns=headers)
```

```python

                df['u_bar'] = df.mean(axis=1)
                for i in range(len(self.data_arr)):
                    df[f'u_prime_{i+1}'] = df['u_bar'] - df[f'u_{i+1}']
                return df

        else:
            for property, in zip(properties):
                for i, temporal_data, in enumerate(self.data_arr):
                    if property == 'u':
                        treated.append(list(temporal_data.u))
                    headers.append(f'{property}_{i+1}')

            df = pd.DataFrame(np.transpose(treated),
                columns=headers)

            df['u_bar'] = df.mean(axis=1)
            for i in range(len(self.data_arr)):
                df[f'u_prime_{i+1}'] = df['u_bar'] - df[f'u_{i+1}']
            return df


if __name__ == '__main__':
    pass
```

A.2 Definição de Funções de Cálculos Numéricos

Código A.2 – Código em Python para a declaração de classes

```python
from statistics import variance
import numpy as np
from matplotlib import pyplot as plt

def temporal_mean(phi:np.ndarray):

    N = len(phi)
    phi_bar = 0

    for i in range(N):
        phi_bar +=  phi[i-1]

    phi_bar /= i

    return phi_bar

def statistical_mean(data_list):

    spacial_averages = np.zeros(len(data_list[0].u))

    for j in range(len(spacial_averages)):
```

```
        spacial_avg = 0.
        for i, data in enumerate(data_list):
            spacial_avg += data.u[j]
            spacial_averages[j] = spacial_avg / len(data_list)

    return spacial_averages


def moving_average(arr, window):
    i = 0
    moving_average = []

    while i < len(arr) - window + 1:

        window_avg = np.sum(arr[i:i+window])/window

        moving_average.append(window_avg)

        i += 1

    return moving_average

def calculate_fluctuation(u:np.ndarray, u_bar:float):

    return u - u_bar


def calculate_ordered_moment(phi_prime:np.ndarray, order: int):

    variance = temporal_mean(phi_prime ** order)

    return variance

def calculate_kinetic_energy(u:np.ndarray):

    kinetic = 0.5 * calculate_ordered_moment(u, 2)

    return kinetic

def calculate_std_dev(phi_prime:np.ndarray):

    variance = calculate_ordered_moment(phi_prime, 2)

    return np.sqrt(variance)

def calculate_turbulence_intensity(u_bar:float, u_prime:np.ndarray):

    phi_rms = calculate_std_dev(u_prime)

    return phi_rms / np.abs(u_bar)

```

```python
def calculate_dissimetry_coef(phi_prime:np.ndarray):

    sigma_3 = calculate_ordered_moment(phi_prime, 3)
    sigma_2 = calculate_ordered_moment(phi_prime, 2)

    return sigma_3 / ((sigma_2)**(3/2))

def calculate_flatenning_coef(phi_prime:np.ndarray):

    sigma_4 = calculate_ordered_moment(phi_prime, 4)
    sigma_2 = calculate_ordered_moment(phi_prime, 2)

    return sigma_4 / (sigma_2**2)

def pdf(u_t, N=None):
    """Probability Density Function for the values of an
    1D array

    Args:
        u_t (np.ndarray): 1D array
        N (int): numbers of intervals

    Returns:
        tuple: tuple of interval and probabitlity density pair
    """
    N_u = len(u_t)
    if N is None:
        N = N_u

    prob = np.zeros(N)
    u = np.sort(u_t)
    x = np.linspace(min(u), max(u), N)
    print('Calculating pdf...')
    var = variance(u)
    pdf = pdf_algorithm(u, x, prob, N, N_u, var, (1/N))
    print('Done alculating pdf.')

    return (x, pdf)

def pdf_algorithm(u, x, pdf, N, N_u, var, TOL):
    """Main probability density function algorithm
    able to work with numba jit method

    This algorithm takes an ordered 1D velocity list
    and calculates the probability of each velocity
    at each point in time.

    Args:
        u (np.ndarray): velocity 1D array
        x (np.ndarray): array with the intervals
        pdf (np.ndarray): empty result array
        N (np.ndarray): number of intervals to be analysed
```

```
        N_u (np.ndarray): number of elements in the velocity array
        TOL (float): a tolerance value to control floating points

    Returns:
        pdf (np.ndarray) : the probability density function array
           correspondent
        to the velocity array
    """
    k = 0
    i = 0
    p = 0
    while k < N_u and i < N:
        if u[k] > x[i] - 0.1*TOL and u[k] < x[i+1]+0.1*TOL:
            k += 1
            p += 1
        else:
            pdf[i] = p/(N_u * var)
            p = 0
            i += 1

    return pdf

def covariance(a_prime:np.ndarray, b_prime:np.ndarray):
    if len(a_prime) != len(b_prime):
        raise ValueError('Matrices must have the same length')

    cov = 0
    for a_p, b_p in zip(a_prime, b_prime):
        cov += a_p * b_p
    cov /= len(a_prime)

    return cov

def correlation_coeff(a_prime:np.ndarray, b_prime:np.ndarray):

    cov_ab = covariance(a_prime,b_prime)
    a_rms = calculate_std_dev(a_prime)
    b_rms = calculate_std_dev(b_prime)
    corr_coef = cov_ab / (a_rms * b_rms)

    return corr_coef
```

A.3 Código para o tratamento de dados

Código A.3 – Código em Python para a declaração de classes

```
from dataclasses import dataclass, field
from statistics import covariance
from matplotlib import pyplot as plt
from scipy.stats import norm
```

```python
import os, os.path
import numpy as np
import pandas as pd
import json
from statistical_functions import *

def export_data(df, path):
    print('Exporting csv ...')
    df.to_csv(f'{path}.csv')
    print('Export complete')

    return df

def get_spatial_points(path):
    points = []
    with open(path, 'r') as f:
        yo = f.readline()[3:-3].split(',')
        yo = float('.'.join(yo))

        data = f.readlines()[1:]
        for line in data:
            line = round(float('.'.join(line[14:-3].split(','))) -
               yo, 1)
            points.append(line)

    return points

def main():
    points_mon = get_spatial_points('data/pos_mon.txt')
    points_jus = get_spatial_points('data/pos_jus.txt')

if __name__ == '__main__':
    main()
```

A.4 Código para rodar o programa

Código A.4 – Código em Python para a declaração de classes

```python
import numpy as np
from data_class import TemporalData, TemporalDatas
from data_man import export_data, get_spatial_points
import os

def run(FOLDERS) -> None:

    for folder in FOLDERS:

        FOLDER = f'data/{folder}/'
        paths = [FOLDER+name for name in os.listdir(FOLDER) if
           os.path.isfile(os.path.join(FOLDER, name))]
```

```
        print(paths)
        N = len(paths)

        data_arr = []

        for i, path in enumerate(paths):
            splitted_path = path.split('/')
            name = splitted_path[-1][:-4]

            if name == 'hre' or name == 'lre':
                index = splitted_path[-1][-5]
            elif name == 'PERFILM':
                index = splitted_path[-1][-2:]
            else:
                index = splitted_path[-1][-6:-4]

            t, u = np.loadtxt(path, unpack=True)
            data = TemporalData(path, name, index, u=u, times=t)
            data.save_txt()

            if name == 'hre' or name == 'lre':
                index = splitted_path[-1][-5]
            elif name == 'PERFILM':
                index = splitted_path[-1][-2:]
            else:
                index = splitted_path[-1][-6:-4]

            data.save_json()
            data_arr.append(data)

        folder = FOLDER.split('/')[-2]
        stat_data = TemporalDatas(name=folder, data_arr=data_arr)

        if folder == 'lre'or folder == 'hre':

            cov_arr = []
            corr_arr = []
            for n in range(N-1):
                cov, corr = stat_data.calculate_cov_corr(n, n+1)
                print(cov, corr)
                cov_arr.append(cov)
                corr_arr.append(corr)
            stat_data.cov = cov_arr
            stat_data.corr_coef = corr_arr

        elif folder == 'perfil_jus' or folder == 'perfil_mon':

            u_bar_arr = []
            for n in range(N):
                u_bar_arr.append(stat_data.data_arr[n].u_bar_t)
                stat_data.u_bar_s = np.array(u_bar_arr)
            print(u_bar_arr)
```

```python
            if folder == 'perfil_jus':
                stat_data.positions = get_spatial_points(f'data/pos_jus.txt')
            elif folder == 'perfil_mon':
                stat_data.positions = get_spatial_points(f'data/pos_mon.txt')
            pass

        print(stat_data.__str__())
        stat_data.save_json()
        df = stat_data.to_data_frame(['u',])
        folder = FOLDER.split('/')[-2]
        export_data(df, f'CSV/{folder}')
        print(df)


    return None


if __name__ == '__main__':
    FOLDERS = ('lre', 'hre', 'hre_prob', 'perfil_mon', 'perfil_jus')
    FOLDERS = ('perfil_mon',)
    run(FOLDERS)
```

A.5 Código para realizar a plotagem dos gráficos

Código A.5 – Código em Python para a declaração de classes

```python
from unicodedata import name
from matplotlib import pyplot as plt, transforms
from scipy.stats import norm
import os
import json
from statistical_functions import pdf
import numpy as np

def plot_instant_velocity(obj):

    dir_name = obj['path'].split("/")[-2]
    save_dir_name = f'images/{dir_name}'

    try:
        os.mkdir(save_dir_name)
    except FileExistsError:
        pass
    fig = plt.figure(f'plot_{obj["name"]}')
    ax = plt.axes()
    try:
        ax.set_title(f'Velocidade instantânea u_{obj["index"]}')
```

```
        ax.plot(obj["times"][:len(obj["u"])], obj["u"])
    except KeyError:
        if obj["name"] == 'hre_prob':
            ax.set_title(f'Velocidade instantânea u_{obj["name"]}')
            ax.plot(obj["times"], obj["u_bar_s"],)


    ax.plot(obj["times"], obj["u_bar_t"]*np.ones(len(obj["times"])), '-r')

    ax.set_xlabel('Tempo [s]')
    ax.set_ylabel('Velocidade [m/s]')

    ax.legend(['Velocidade instantânea', 'Média temporal'])

    plt.savefig(f'{save_dir_name}/instantaneous_velocity_{obj["name"]}.png')
    plt.close(fig)


def plot_pdf(obj):
    dir_name = obj["path"].split("/")[-2]
    save_dir_name = f'images/{dir_name}'

    x, pdf_u = obj["u_x_pdf"], obj["u_pdf"]
    mu = obj["u_bar_t"]
    sigma = np.sqrt(obj["variance"])
    gauss = norm.pdf(x, mu, sigma)

    fig = plt.figure(f'probability_density_function_{obj["name"]}', [6, 6])
    ax = plt.axes()
    ax.set_title(f'Velocity by Probablility Density Function {obj["name"]}')
    ax.grid(1, 'both')

    # ax.plot(x, pdf_u, '-b')
    ax.hist(obj["u"], bins=500, density=True, stacked=True)
    ax.plot(x, gauss, '-r', lw=1.75)
    ax.vlines(obj["u_bar_t"], min(pdf_u), max(gauss), colors='k', linestyles='dashed')

    ax.set_xlabel('Velocity $u$ [m/s]')
    ax.set_ylabel('Probability Density Function $P(u)$')
    plt.savefig(f'{save_dir_name}/pdf_{obj["name"]}.png')
    plt.close(fig)

def plot_pdf_hrep(obj):
    dir_name = obj['path'].split("/")[-2]
    save_dir_name = f'images/{dir_name}'

    try:
```

```
        os.mkdir(save_dir_name)
    except FileExistsError:
        pass

    x, pdf_u = obj["u_x_pdf"], np.array(obj["u_pdf"])*100
    mu = obj["u_bar_t"]
    sigma = obj["u_rms"]
    gauss = norm.pdf(x, mu, sigma)

    fig = plt.figure(f'probability_density_function_{obj["name"]}', [6, 6])
    ax = plt.axes()
    ax.set_title(f'Velocity by Probablility Density Function {obj["name"]}')
    ax.grid(1, 'both')

    # ax.plot(x, pdf_u, '-b')
    plt.hist(obj["u_bar_s"], bins=500, density=True, stacked=True)
    ax.plot(x, gauss, '-r', lw=1.75)
    ax.vlines(obj["u_bar_t"], min(pdf_u), max(gauss), colors='k', linestyles='dashed')

    ax.set_label('Velocity $u$ [m/s]')
    ax.set_ylabel('Probability Density Function $P(u)$ [%]')
    plt.savefig(f'{save_dir_name}/pdf_{obj["name"]}.png')
    plt.close(fig)

def plot_velocity_distribution(obj):
    dir_name = obj['path'].split("/")[-2]
    save_dir_name = f'images/{dir_name}'

    try:
        os.mkdir(save_dir_name)
    except FileExistsError:
        pass

    base = plt.gca().transData
    rot = transforms.Affine2D().rotate_deg(90)

    fig = plt.figure(f'velocity_distribution_{obj["name"]}')
    ax = plt.axes()
    ax.set_title(f'Distribuição de velocidade {obj["name"]}')
    ax.grid(1, 'both')

    ax.plot(obj["u_bar_s"], obj["positions"], '-b')
    # for position in obj["positions"]:
    #     ax.text(obj["u_bar_s"][position], position, obj["u_bar_s"][position], transform=rot+base)
    # ax.vlines(obj["u_bar_t"], min(obj["positions"]), max(obj["positions"]), colors='k', linestyles='dashed')

```

```python
    ax.set_ylabel('Position in y-axis $P_y$')
    ax.set_xlabel('Velocity $u$ [m/s]')
    plt.savefig(f'{save_dir_name}/velocity_distribution_{obj["name"]}.png')
    plt.close(fig)

def main():
    try:
        os.mkdir('images')
    except FileExistsError:
        pass
    FOLDERS = ('lre', 'hre', 'hre_prob', 'perfil_mon',
       'perfil_jus')
    # FOLDERS = ('perfil_mon', 'perfil_jus')
    for folder in FOLDERS:

        FOLDER = f'json_results/{folder}/'
        paths = [FOLDER+name for name in os.listdir(FOLDER) if
           os.path.isfile(os.path.join(FOLDER, name))]
        for path in paths:
            print(f'{path}')
            try:
                os.mkdir(f'images/{folder}')
            except FileExistsError:
                pass
            with open(path) as data:
                obj = json.load(data)
            if ((folder == 'lre' or folder == 'hre')
                and (path.split("/")[-1] != 'lre.json' and
                   path.split("/")[-1] != 'hre.json')):
                plot_instant_velocity(obj)
                plot_pdf(obj)
            elif folder == 'hre_prob':
                if path.split("/")[-1] == 'hre_prob.json':
                    plot_pdf_hrep(obj)
                else:
                    plot_instant_velocity(obj)
                    plot_pdf(obj)
            elif path.split("/")[-1] == 'perfil_mon.json' or
               path.split("/")[-1] == 'perfil_jus.json':
                plot_velocity_distribution(obj)

if __name__ == '__main__':
    main()
```