Lista_tarefas

October 22, 2025

1 Tarefa 1

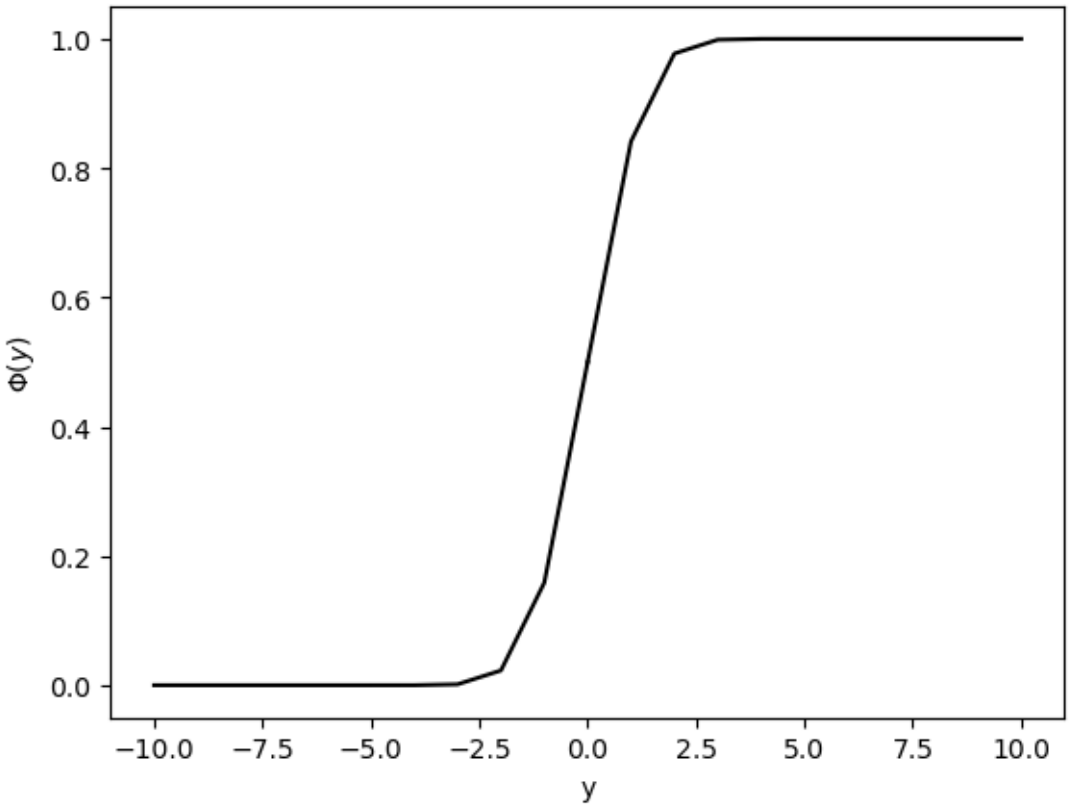
A proposta é implementar a solução aproximada para a a função de probabilidade acumulada normal padrão $\Phi(y)$, de acordo com o apresentado no Anexo F do livro de referência. O código abaixo apresenta a construção das aproximações de $\Phi(y)$ para os intervalos $0 \leq y \leq \infty$ e $-\infty \leq y \leq 0$

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# Valores do parâmetro p_i
p_i = [0.231641900, 0.319381530, -0.356563782, 1.781477937, -1.821255978, 1.
 ↪330274429]

# Função w
def w (y):
    return 1 / (1 + (p_i[0] * abs(y)))

# Função z
def z (w):
    return (w * (p_i[1] + w * (p_i[2] + w * (p_i[3] + w * (p_i[4] + w *␣
 ↪p_i[5])))))

# Aproximação analítica para a função de probabilidade acumulada
# Para y negativo
def phi_neg (z , y):
    return (z / np.sqrt( 2 * np.pi)) * np.exp(- (y**2 / 2))

# Para y positivo
def phi_pos (z , y):
    return 1 - (z / np.sqrt( 2 * np.pi)) * np.exp(- (y**2 / 2))
```

O código abaixo apresenta a verificação e plotagem da função $\Phi(y)$ para o intervalo $-10 \leq y \leq 10$

[6]:
```
# Verificação das funções
# Vetores para guardar os resultados
phi_neg_results = []
phi_pos_results = []
y_neg = []
```

```
y_pos = []
for y in range(-10 , 11, 1):
    if y < 0:
        w_calc = w(y)
        z_calc = z(w_calc)
        phi_neg_calc = phi_neg(z_calc, y)
        phi_neg_results.append(phi_neg_calc)
        y_neg.append(y)
    elif y == 0:
        w_calc = w(y)
        z_calc = z(w_calc)
        phi_neg_calc = phi_neg(z_calc, y)
        phi_neg_results.append(phi_neg_calc)
        y_neg.append(y)
        phi_pos_calc = phi_pos(z_calc, y)
        phi_pos_results.append(phi_pos_calc)
        y_pos.append(y)
    else:
        w_calc = w(y)
        z_calc = z(w_calc)
        phi_pos_calc = phi_pos(z_calc, y)
        phi_pos_results.append(phi_pos_calc)
        y_pos.append(y)

# Plotagem dos resultados
fig, ax = plt.subplots()
ax.plot(y_neg, phi_neg_results, color='black')
ax.plot(y_pos, phi_pos_results, color='black')
plt.xlabel('y')
plt.ylabel('$\\Phi(y)$')
plt.show()
```



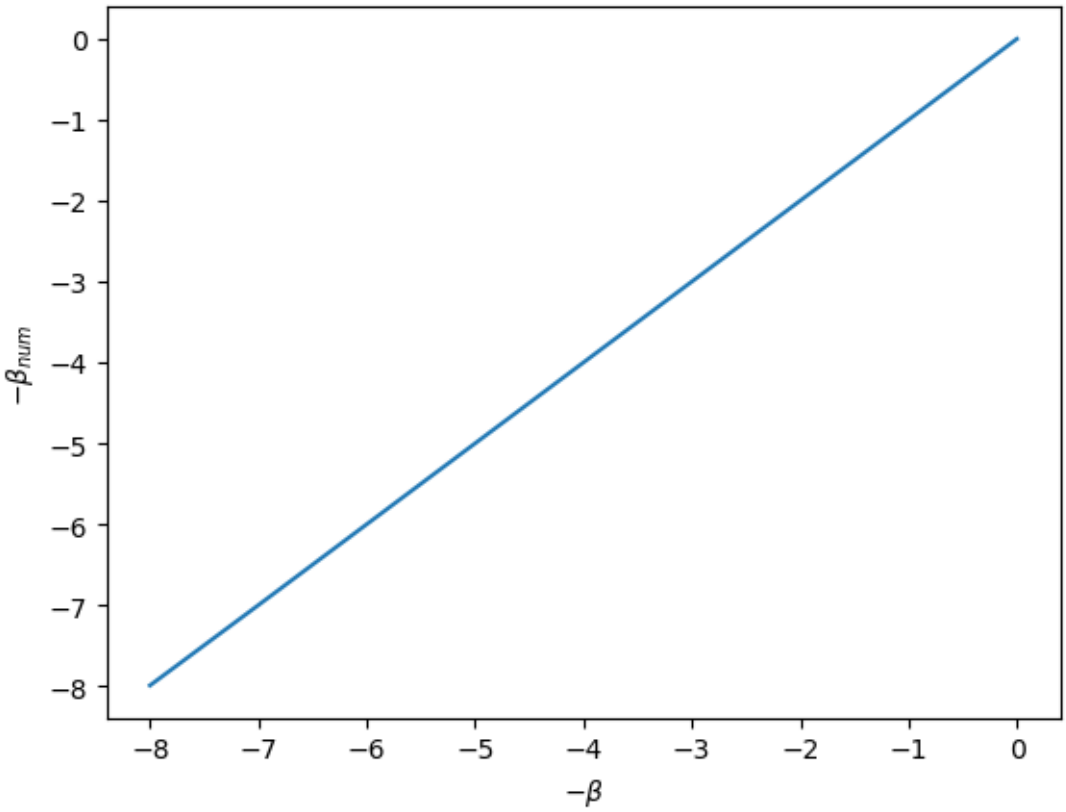
Agora temos que o código abaixo apresenta a implementação da função CDF inversa $y = \Phi^{-1}(u)$

[7]:
```
# Formulação da função inversa

# Valores do parâmetro p_i
p = [-0.3222324310880, -1.0000000000000, -0.3422422088547, -0.2042312102450e-1,␣
 ↪-0.4536422101480e-4]

# Valores do parâmetro q_i
q = [0.9934846260600e-1, 0.5885815704950, 0.5311034623660, 0.10353775285000, 0.
 ↪3856070063400e-2]

# Função inversa
# Para 0 < u <= 0.5
def y_1 (u):
    z = np.sqrt(np.log(1 / (u ** 2)))
    return -z - ((p[0] + z * (p[1] + z * (p[2] + z * (p[3] + z * (p[4]))))) /␣
 ↪(q[0] + z * (q[1] + z * (q[2] + z * (q[3] + z * (q[4]))))))

# Para 0.5 <= u < 1
def y_2 (u):
```

```
    z = np.sqrt(np.log (1 / ((1 - u) ** 2)))
    return z + ((p[0] + z * (p[1] + z * (p[2] + z * (p[3] + z * (p[4]))))) /␣
↪(q[0] + z * (q[1] + z * (q[2] + z * (q[3] + z * (q[4]))))))
```

Como verificação da implementação, calcula-se $B_{num} = \Phi^{-1}(\Phi(-B))$. O código abaixo apresenta o cálculo de B_{num} e plota o resultado para um intervalo $-8 \leq B \leq 0$

[8]:
```
# Verificação da implementação

vetor_beta = []
vetor_beta_aprox = []

for i in np.arange (0, 9, 1):
    w_calc = w(i)
    z_calc = z(w_calc)
    vetor_beta.append(-1 * i)
    result = 1- phi_pos(z_calc, i)
    if result > 0:
        if result <= 0.5:
            inverse_result = y_1(result)
        else:
            inverse_result = y_2(result)
        vetor_beta_aprox.append(inverse_result)

plt.plot(vetor_beta, vetor_beta_aprox)
plt.xlabel('$-\\beta$')
plt.ylabel('$-\\beta_{num}$')
plt.show()
```



2 Tarefa 2

Temos que no código abaixo possui a implementação das seguintes distribuições: - Distribuição normal; - Distribuição log-normal; - Gumbel para máximos e Gumbel para mínimos

No código abaixo, temos uma estrutura de *data members*, na qual é possível calcular os momentos da distribuição (média, variância, desvio-padrão, coeficiente de variação, skewness e kurtosis) dado os parâmetros. Além disso, há uma estrutura de *member functions* que calcula as funções PDF, CDF e CDF^{-1} e também calcula os parâmetros dados os momentos.

[1]:
```
from scipy import stats as st
import numpy as np
import matplotlib.pyplot as plt

# Data members
class variavel_aleatoria:
    # Função para identificar qual é a distribuição, o nome da variável e o␣
 ↪simbolo da distribuição
    def __init__(self, distribuicao: str, nome: str = "", simbolo: str =""):
        # Identificação
        self.nome = nome
```

```
        self.simbolo = simbolo
        self.distribuicao = distribuicao

        # Lista de argumentos
        self.parametros = []
        self.objeto = None

        # Momentos da variável
        self.media = np.nan
        self.variancia = np.nan
        self.desvio = np.nan
        self.cv = np.nan # Coeficiente de variação
        self.skewness = np.nan
        self.kurtosis = np.nan

        # Distribuições implementadas
        self.distribuicoes = {
            'normal' : st.norm,
            'lognormal' : st.lognorm,
            'gumbel_max': st.gumbel_r,
            'gumbel_min': st.gumbel_l,
        }

        # Aqui as distribuições contempladas são atribuidas ao componente objeto
        self.objeto = self.distribuicoes[self.distribuicao]

        # Aqui os parâmetros de cada distribuição são definidos e os momentos␣
↪calculados a partir dos parâmetros
    def conjunto_parametros (self, *params):
        self.parametros = list(params)
        self.calculo_momentos()

        # Aqui os momentos são calculados a partir dos parametros
    def calculo_momentos(self):
        m, v, sk, k = self.objeto.stats(*self.parametros, moments = 'mvsk')

        # Armazenamento dos momentos nas variáveis
        self.media = float(m)
        self.variancia = float(v)
        self.desvio = np.sqrt(self.variancia)
        self.skewness = float(sk)
        self.kurtosis = float(k)

        if self.media != 0:
            self.cv = self.desvio / self.media
        else:
            self.cv = np.nan
```

```
    # Aqui calcula-se os parametros de cada distribuição dado os momentos␣
↪(média e desvio padrão)
    def calculo_parametros (self, media_dada: float, desvio_dado: float):
        mu = media_dada
        sigma = desvio_dado

        if self.distribuicao =='normal':
            self.conjunto_parametros(mu, sigma)

        elif self.distribuicao == 'lognormal':
            zeta = np.sqrt(np.log(1.0 + (sigma / mu) ** 2))
            lam = np.log(mu) - (0.5 * (zeta ** 2))
            scale = np.exp(lam)
            self.conjunto_parametros(zeta, 0.0, scale)

        elif self.distribuicao in ['gumbel_max', 'gumbel_min']:
            mu = media_dada
            sigma = desvio_dado
            gamma = 0.5772156649 #Constante de Euler
            beta = (sigma * np.sqrt(6)) / np.pi

            if self.distribuicao == 'gumbel_max':
                mu_calc = mu - (gamma * beta)
            else:
                mu_calc = mu + (gamma * beta)
            self.conjunto_parametros(mu_calc, beta)

    # Agora vamos construir as funções fundamentais (PDF, CDF, Inversa)
    def PDF (self, x: float) -> float:
        if self.objeto:
            return self.objeto.pdf(x, *self.parametros)
        return np.nan

    def CDF (self, x: float) -> float:
        if self.objeto:
            return self.objeto.cdf(x, *self.parametros)
        return np.nan

    def InversaCDF (self, p: float) -> float:
        if self.objeto:
            return self.objeto.ppf(p, *self.parametros)
        return np.nan
```

```
/home/gabrielsilverio/anaconda3/envs/confiabilidade_env/lib/python3.13/site-
packages/numpy/_core/getlimits.py:552: UserWarning: Signature
b'\x00\xd0\xcc\xcc\xcc\xcc\xcc\xcc\xfb\xbf\x00\x00\x00\x00\x00\x00' for <class
'numpy.longdouble'> does not match any known type: falling back to type probe
```

```
function.
This warnings indicates broken support for the dtype!
  machar = _get_machar(dtype)
```

Agora, para cada distribuição implementada, vamos testar a estrutura anterior calculando os momentos das variáveis dado os parâmetros e plotando as funções PDF e $x_{aprox} = CDF^{-1}(x, CDF(X, x))$.

3 Teste da estrutura para a distribuição normal

Neste teste vamos supor uma variável X ~ N(50,200) e apresentar o cálculo dos momentos à partir dos parâmetros.

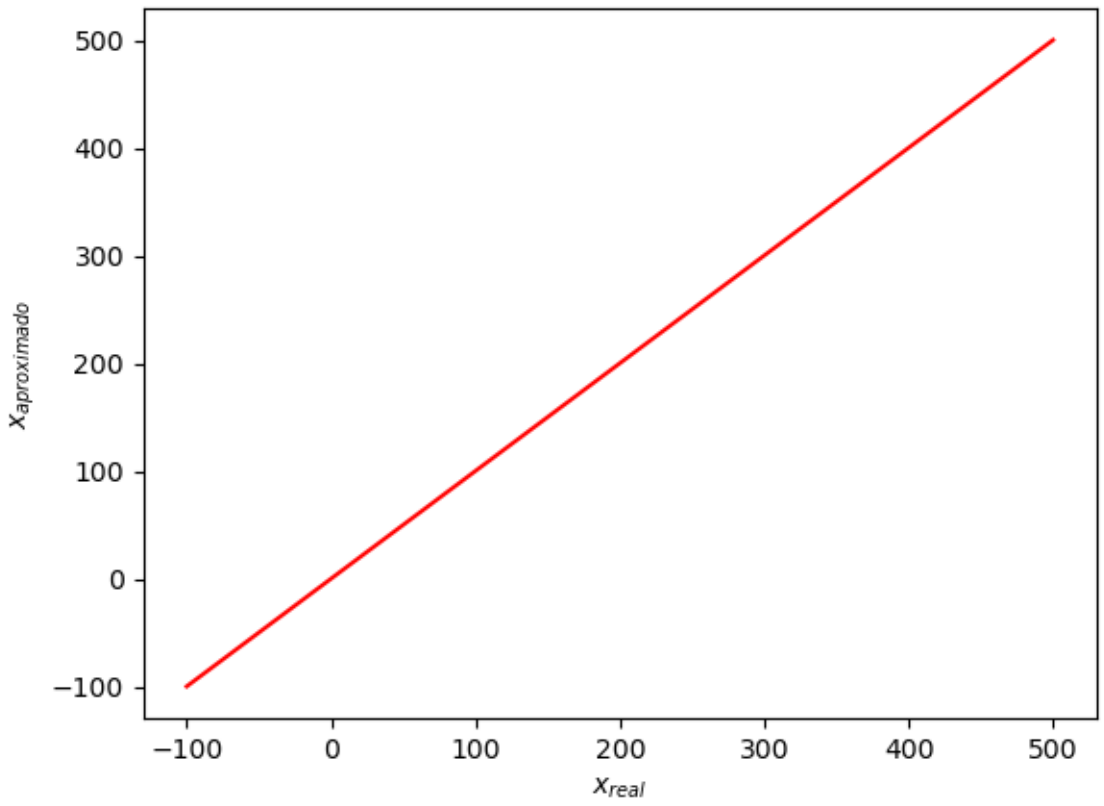
[48]:
```
#Teste da estrutura
X_normal = variavel_aleatoria(distribuicao = 'normal', nome='VA normal',␣
 ↪simbolo='X_N')
media_dada = 200
sigma_dado = 50
X_normal.conjunto_parametros(media_dada, sigma_dado)

# Teste do conjunto de parametros e momentos para uma distribuição normal
print('Verificação da distribuição normal: Calculo dos momentos dados os␣
 ↪parâmetros')
print(f"X: {X_normal.nome}")
print(f"Parâmetros: {X_normal.parametros}")
print(f"Média calculada: {X_normal.media}")
print(f"Desvio padrão: {X_normal.desvio}")
print(f"Coeficiente de variação: {X_normal.cv}")
print(f"Skewness: {X_normal.skewness}")
print(f"Kurtosis: {X_normal.kurtosis}")
```

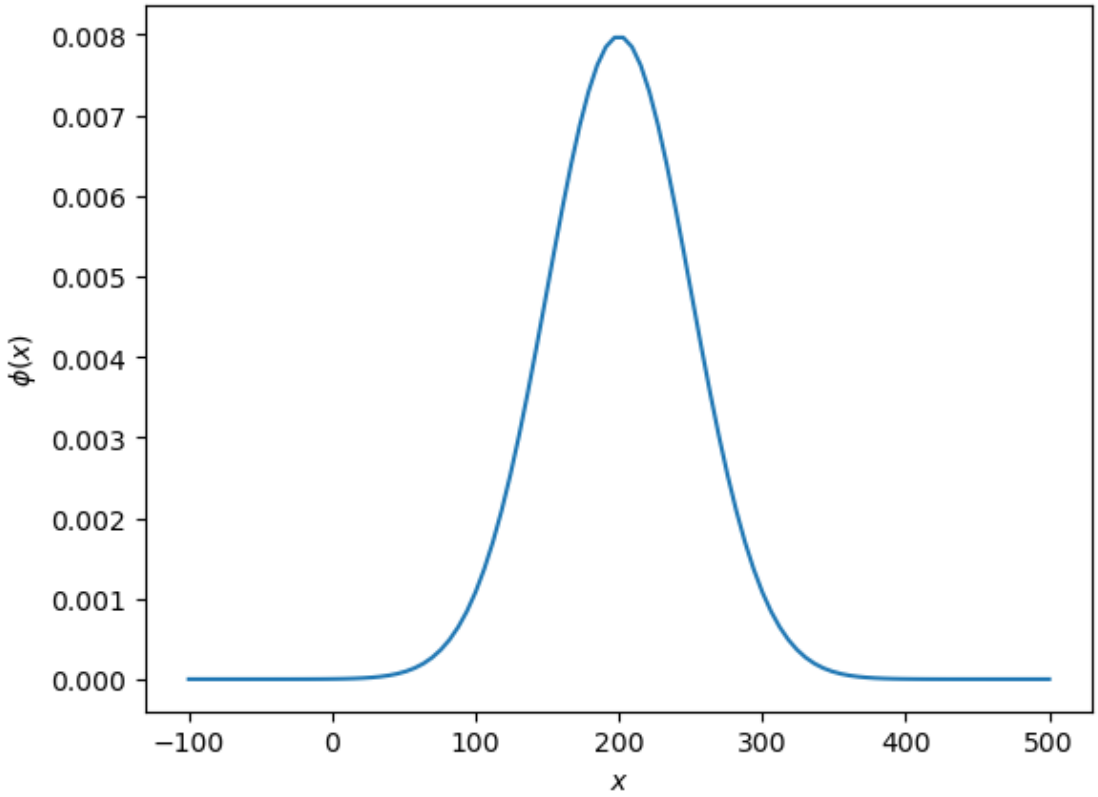
```
Verificação da distribuição normal: Calculo dos momentos dados os parâmetros
X: VA normal
Parâmetros: (200, 50)
Média calculada: 200.0
Desvio padrão: 50.0
Coeficiente de variação: 0.25
Skewness: 0.0
Kurtosis: 0.0
```

Uma vez calculados os parâmetros, agora vamos vamos plotar os resultados $x_{aprox} = CDF^{-1}(x, CDF(X, x))$ para um intervalo $\mu - 3\sigma \leq x \leq \mu + 3\sigma$ e também vamos plotar a função PDF.

[49]:
```
import matplotlib.pyplot as plt

x_min = media_dada - (6 * sigma_dado)
x_max = media_dada + (6 * sigma_dado)
```

```
x_real = np.linspace(x_min, x_max, 100)

x_aproximado = []
pdf = []
for x in x_real:
    p = X_normal.CDF(x)
    pdf_calc = X_normal.PDF(x)
    x_calc = X_normal.InversaCDF(p)
    x_aproximado.append(x_calc)
    pdf.append(pdf_calc)

# Gráfico do valor aproximado x valor real
plt.plot(x_real, x_aproximado, 'r')
plt.xlabel('$x_{real}$')
plt.ylabel('$x_{aproximado}$')
plt.show()

# Plotar função PDF
plt.plot(x_real, pdf)
plt.xlabel('$x$')
plt.ylabel('$\\phi(x)$')
plt.show()
```





4 Teste da estrutura para a distribuição lognormal

Neste teste vamos supor uma variável X ~ LN(5, 0.2) e apresentar o cálculo dos momentos à partir dos parâmetros.

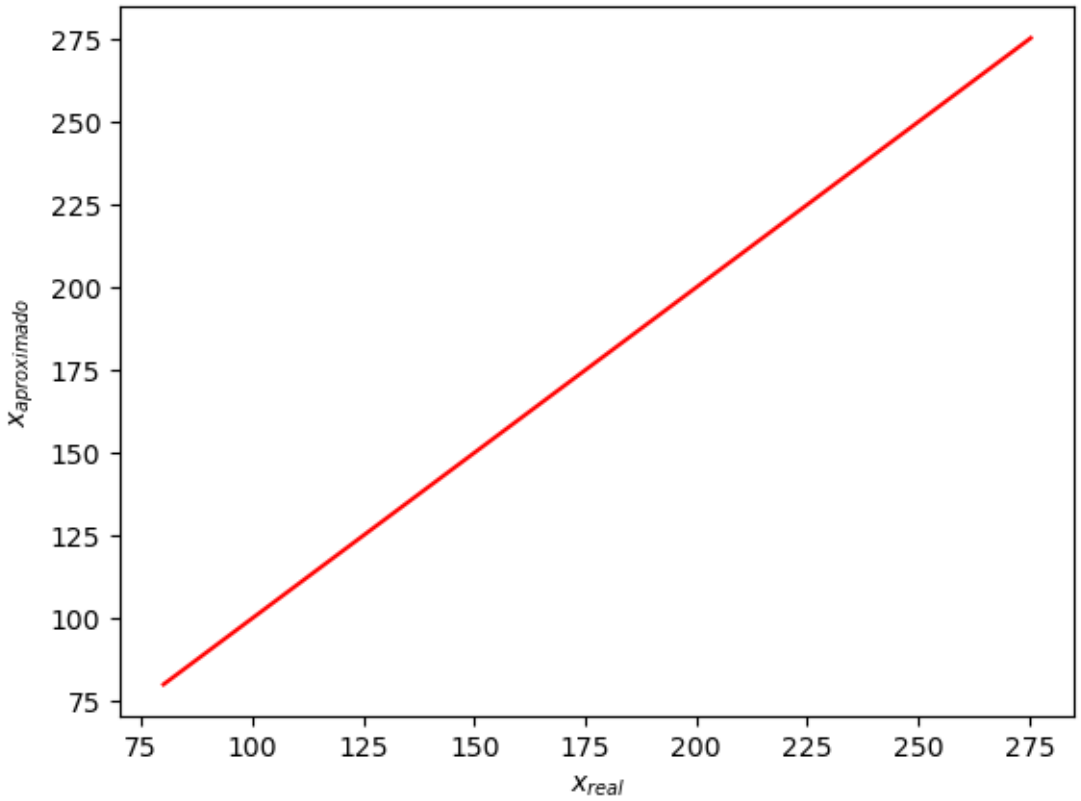
[42]:
```
#Teste da estrutura
X_lognormal = variavel_aleatoria(distribuicao = 'lognormal', nome='VA␣
 ↪lognormal', simbolo='X_LN')
zeta_dado = 0.2
lamb_dado = 5
X_lognormal.conjunto_parametros(zeta_dado, 0, float(np.exp(lamb_dado)))

# Teste do conjunto de parametros e momentos para uma distribuição normal
print('Verificação da distribuição lognormal: Calculo dos momentos dados os␣
 ↪parâmetros')
print(f"X: {X_lognormal.nome}")
print(f"Parâmetros: {lamb_dado, zeta_dado}")
print(f"Média calculada: {X_lognormal.media}")
print(f"Desvio padrão: {X_lognormal.desvio}")
```

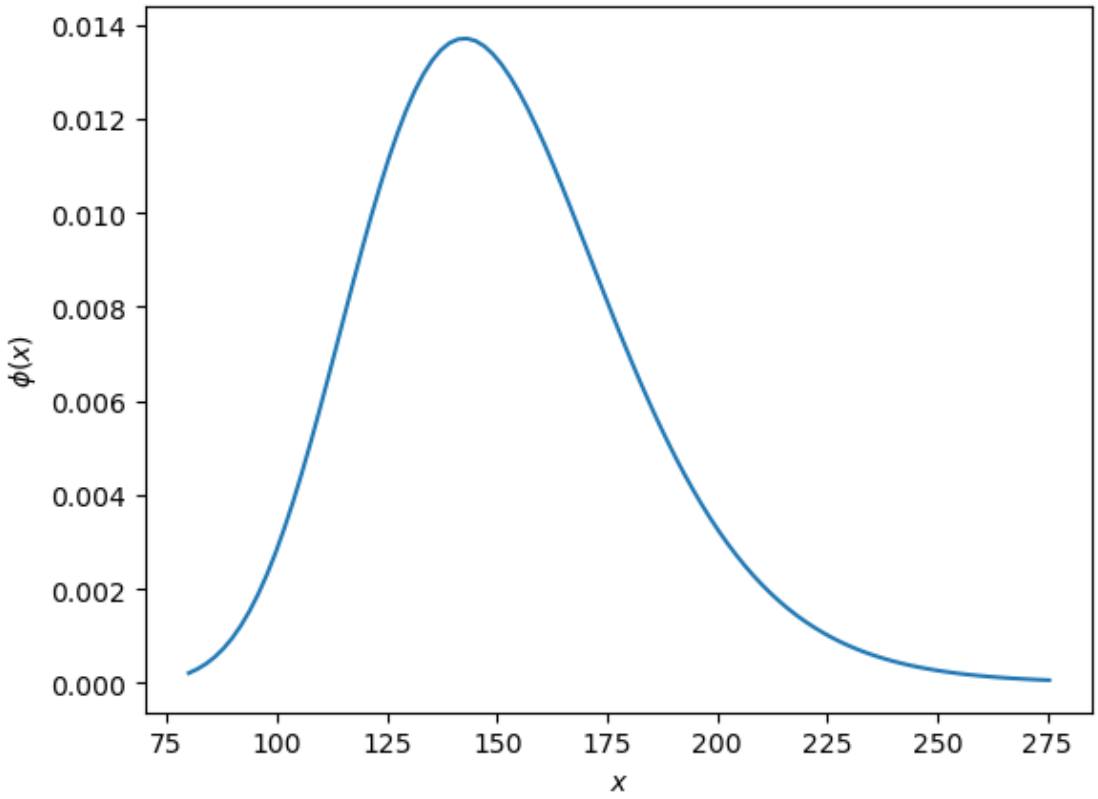
```
print(f"Coeficiente de variação: {X_lognormal.cv}")
print(f"Skewness: {X_lognormal.skewness}")
print(f"Kurtosis: {X_lognormal.kurtosis}")
```

```
Verificação da distribuição lognormal: Calculo dos momentos dados os parâmetros
X: VA lognormal
Parâmetros: (5, 0.2)
Média calculada: 151.41130379405274
Desvio padrão: 30.587622095939498
Coeficiente de variação: 0.20201676710706024
Skewness: 0.6142947619866632
Kurtosis: 0.6783657771754372
```

Uma vez calculados os parâmetros, agora vamos vamos plotar os resultados $x_{aprox} = CDF^{-1}(x, CDF(X, x))$ e também vamos plotar a função PDF.

[46]:
```
import matplotlib.pyplot as plt

x_min = 0.001
x_max = 0.999
x_min_plot = X_lognormal.InversaCDF(x_min)
x_max_plot = X_lognormal.InversaCDF(x_max)
x_real = np.linspace(x_min_plot, x_max_plot, 100)

x_aproximado = []
pdf = []
for x in x_real:
    p = X_lognormal.CDF(x)
    pdf_calc = X_lognormal.PDF(x)
    x_calc = X_lognormal.InversaCDF(p)
    x_aproximado.append(x_calc)
    pdf.append(pdf_calc)

# Gráfico do valor aproximado x valor real
plt.plot(x_real, x_aproximado, 'r')
plt.xlabel('$x_{real}$')
plt.ylabel('$x_{aproximado}$')
plt.show()

# Plotar função PDF
plt.plot(x_real, pdf)
plt.xlabel('$x$')
plt.ylabel('$\\phi(x)$')
plt.show()
```


275
250
225
200
175
150
125
100
75
$x_{aproximado}$
75
100
125
150
175
200
225
250
275
x_{real}



5 Teste da estrutura para a distribuição Gumbel para máximos

Neste teste vamos supor uma variável EVI ~ LN(5, 0.1) e apresentar o cálculo dos momentos à partir dos parâmetros.

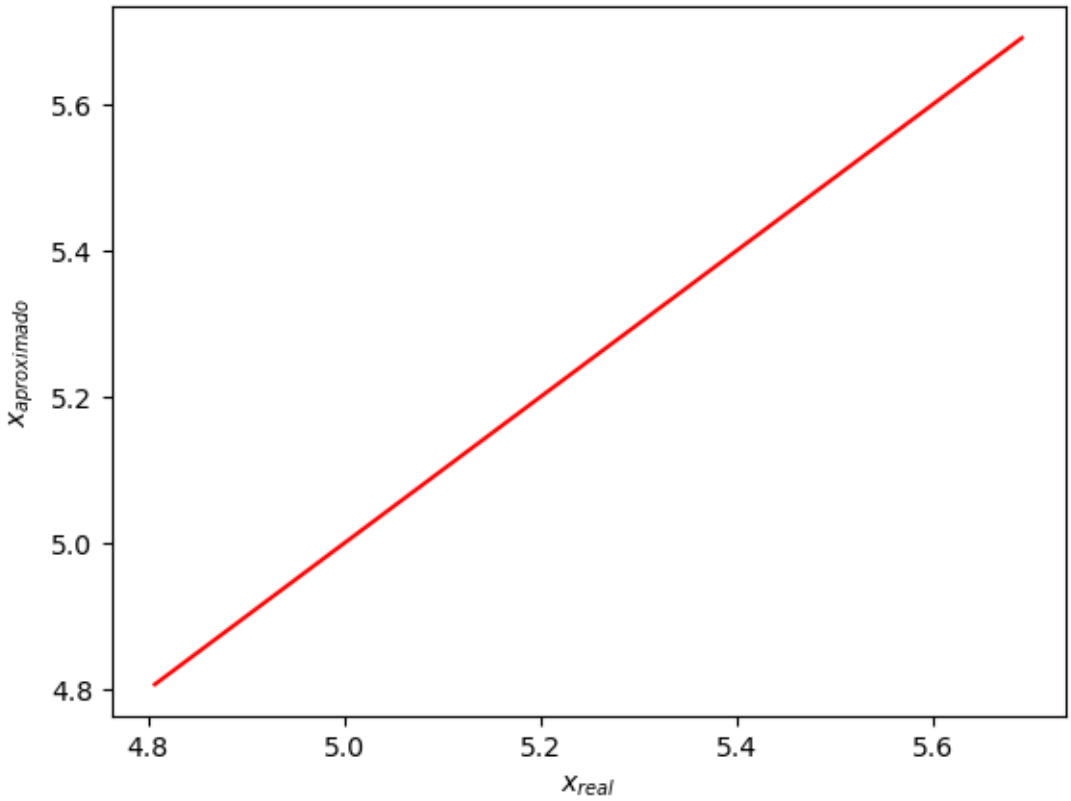
[57]:
```
#Teste da estrutura
X_gumbel_max = variavel_aleatoria(distribuicao = 'gumbel_max', nome='VA Gumbel␣
 ↪para máximos', simbolo='X_EVI')
mi_dado = 5
beta_dado = 0.1
X_gumbel_max.conjunto_parametros(mi_dado, beta_dado)

# Teste do conjunto de parametros e momentos para uma distribuição normal
print('Verificação da distribuição Gumbel para máximos: Calculo dos momentos␣
 ↪dados os parâmetros')
print(f"X: {X_gumbel_max.nome}")
print(f"Parâmetros: {mi_dado, beta_dado}")
print(f"Média calculada: {X_gumbel_max.media}")
print(f"Desvio padrão: {X_gumbel_max.desvio}")
print(f"Coeficiente de variação: {X_gumbel_max.cv}")
```

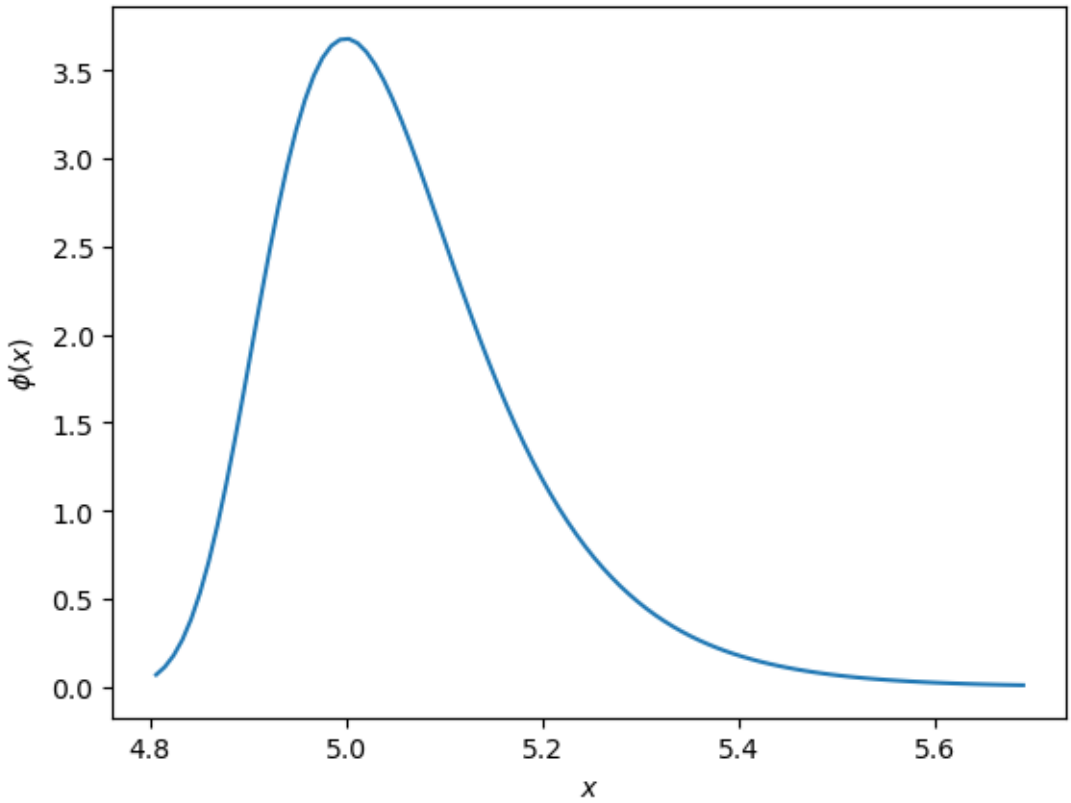
```
print(f"Skewness: {X_gumbel_max.skewness}")
print(f"Kurtosis: {X_gumbel_max.kurtosis}")
```

```
Verificação da distribuição Gumbel para máximos: Calculo dos momentos dados os
parâmetros
X: VA Gumbel para máximos
Parâmetros: (5, 0.1)
Média calculada: 5.057721566490153
Desvio padrão: 0.1282549830161864
Coeficiente de variação: 0.025358252986075306
Skewness: 1.1395470994046486
Kurtosis: 2.4
```

Uma vez calculados os parâmetros, agora vamos vamos plotar os resultados $x_{aprox} = CDF^{-1}(x, CDF(X, x))$ e também vamos plotar a função PDF.

[58]:
```
import matplotlib.pyplot as plt

x_min = 0.001
x_max = 0.999
x_min_plot = X_gumbel_max.InversaCDF(x_min)
x_max_plot = X_gumbel_max.InversaCDF(x_max)
x_real = np.linspace(x_min_plot, x_max_plot, 100)

x_aproximado = []
pdf = []
for x in x_real:
    p = X_gumbel_max.CDF(x)
    pdf_calc = X_gumbel_max.PDF(x)
    x_calc = X_gumbel_max.InversaCDF(p)
    x_aproximado.append(x_calc)
    pdf.append(pdf_calc)

# Gráfico do valor aproximado x valor real
plt.plot(x_real, x_aproximado, 'r')
plt.xlabel('$x_{real}$')
plt.ylabel('$x_{aproximado}$')
plt.show()

# Plotar função PDF
plt.plot(x_real, pdf)
plt.xlabel('$x$')
plt.ylabel('$\\phi(x)$')
plt.show()
```


5.6
5.4
5.2
5.0
4.8
$x_{aproximado}$
4.8
5.0
5.2
5.4
5.6
x_{real}



6 Teste da estrutura para a distribuição Gumbel para mínimo

Neste teste vamos supor uma variável EVI ~ LN(5, 0.1) e apresentar o cálculo dos momentos à partir dos parâmetros.

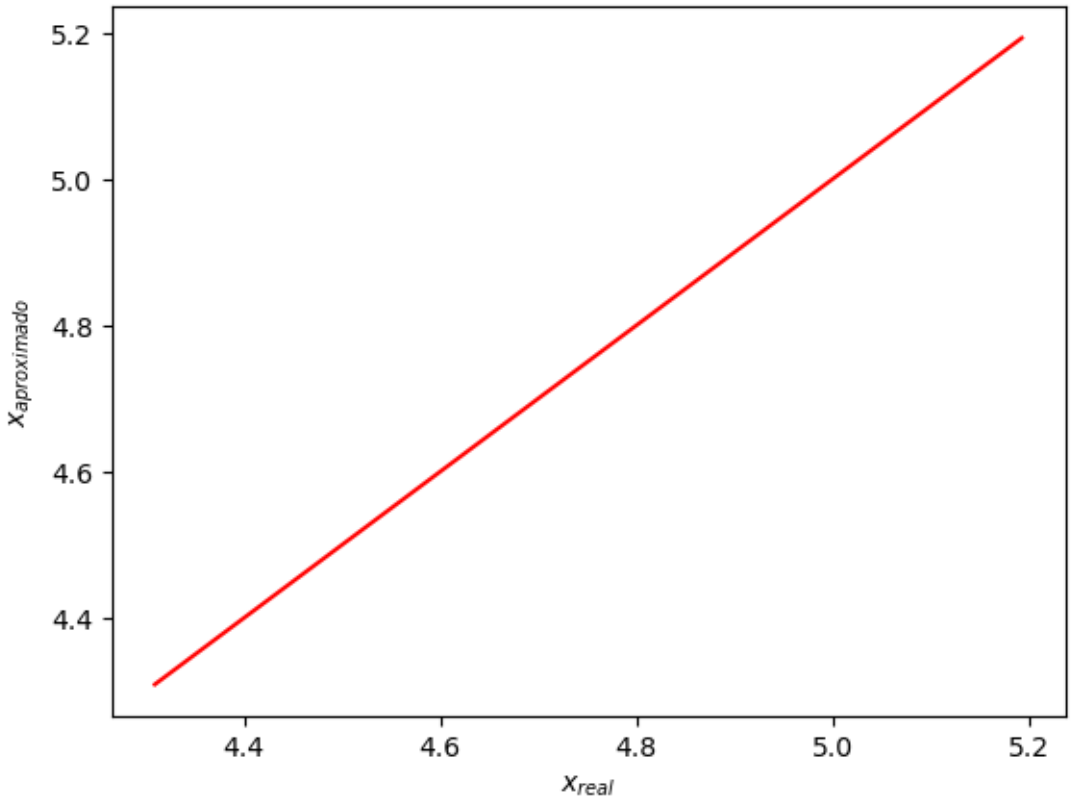
[62]:
```
#Teste da estrutura
X_gumbel_min = variavel_aleatoria(distribuicao = 'gumbel_min', nome='VA Gumbel␣
 ↪para mínimos', simbolo='X_EVI')
mi_dado = 5
beta_dado = 0.1
X_gumbel_min.conjunto_parametros(mi_dado, beta_dado)

# Teste do conjunto de parametros e momentos para uma distribuição normal
print('Verificação da distribuição Gumbel para mínimos: Calculo dos momentos␣
 ↪dados os parâmetros')
print(f"X: {X_gumbel_min.nome}")
print(f"Parâmetros: {mi_dado, beta_dado}")
print(f"Média calculada: {X_gumbel_min.media}")
print(f"Desvio padrão: {X_gumbel_min.desvio}")
print(f"Coeficiente de variação: {X_gumbel_min.cv}")
```

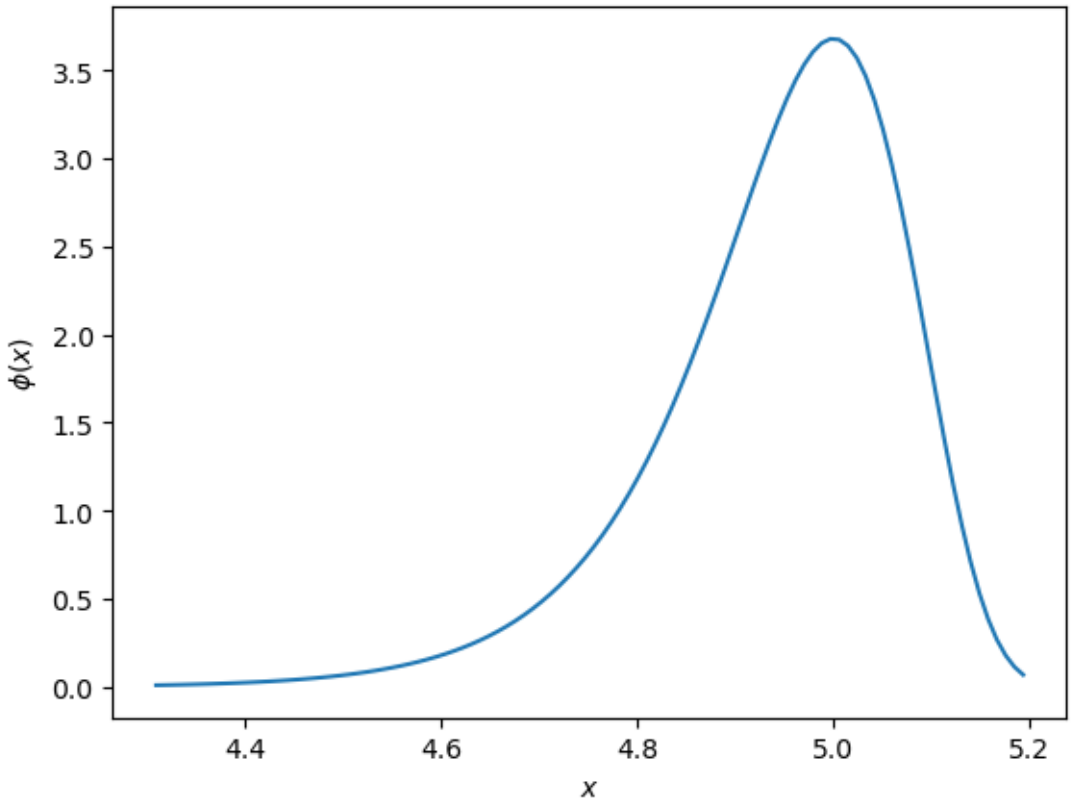
```
print(f"Skewness: {X_gumbel_min.skewness}")
print(f"Kurtosis: {X_gumbel_min.kurtosis}")
```

```
Verificação da distribuição Gumbel para mínimos: Calculo dos momentos dados os
parâmetros
X: VA Gumbel para mínimos
Parâmetros: (5, 0.1)
Média calculada: 4.942278433509847
Desvio padrão: 0.1282549830161864
Coeficiente de variação: 0.025950578208339396
Skewness: -1.1395470994046486
Kurtosis: 2.4
```

Uma vez calculados os parâmetros, agora vamos vamos plotar os resultados $x_{aprox} = CDF^{-1}(x, CDF(X, x))$ e também vamos plotar a função PDF.

[63]:
```
import matplotlib.pyplot as plt

x_min = 0.001
x_max = 0.999
x_min_plot = X_gumbel_min.InversaCDF(x_min)
x_max_plot = X_gumbel_min.InversaCDF(x_max)
x_real = np.linspace(x_min_plot, x_max_plot, 100)

x_aproximado = []
pdf = []
for x in x_real:
    p = X_gumbel_min.CDF(x)
    pdf_calc = X_gumbel_min.PDF(x)
    x_calc = X_gumbel_min.InversaCDF(p)
    x_aproximado.append(x_calc)
    pdf.append(pdf_calc)

# Gráfico do valor aproximado x valor real
plt.plot(x_real, x_aproximado, 'r')
plt.xlabel('$x_{real}$')
plt.ylabel('$x_{aproximado}$')
plt.show()

# Plotar função PDF
plt.plot(x_real, pdf)
plt.xlabel('$x$')
plt.ylabel('$\\phi(x)$')
plt.show()
```


5.2
5.0
4.8
4.6
4.4
$x_{aproximado}$
4.4
4.6
4.8
5.0
5.2
x_{real}



7 TAREFA 3

Temos que o código abaixo faz o caclulo da matriz de correlação R_z utilizando a distribuição de Nataf e calcula os Jacobianos J_{yz} e J_{zy} da tranformação $(Z->Y)$ tanto utilizando a decomposição de Cholesky quanto a decomposição ortogonal.

[ ]:
```
import numpy as np
from UQpy.transformations import Nataf
from UQpy.distributions import Normal as uqpynormal
from UQpy.distributions import Lognormal as uqpylognormal
from UQpy.distributions.collection import GeneralizedExtreme as uqpygev

# Função de mapeamento para o uso do UQpy que retorna os parametros de cada␣
 ↪distribuição
def mapeamento_uqpy(va_cust):
    tipo = va_cust.distribuicao.lower()
    parametros = va_cust.parametros

    if tipo == 'normal':
        return uqpynormal(loc=parametros[0], scale=parametros[1])
```

```
    elif tipo == 'lognormal':
        return uqpylognormal(s=parametros[0], loc=parametros[1],␣
 ↪scale=parametros[2])

    elif tipo in ['gumbel_max', 'gumbel_min']:
        shape_c = 0.0
        loc = parametros[0]
        scale = parametros[1]
        return uqpygev(c=shape_c, loc=loc, scale=scale)

    else:
        raise ValueError(f"Distribuição '{tipo}' não mapeada")

# Data members
class vetores_variavel_aleatoria:
    matriz_observações: np.ndarray # Cada linha é uma variável aleatória X_i e␣
 ↪cada coluna uma observação
    vetor_va_cust: list
    matriz_correlacao_x: np.ndarray # Matriz de correlação
    matriz_correlacao_z: np.ndarray = None

    # Função de recebimento do vetor
    def __init__(self, matriz_observacoes: np.ndarray, vetor_va_cust: list):

        self.matriz_observações = matriz_observacoes
        self.vetor_va_cust = vetor_va_cust
        self.calc_matriz_correlacao()

    # Determinação da dimensão do vetor de variáveis aleatórias
    def dimensao (self) -> tuple:
        return np.shape(self.vetor_va_cust)

    # Calculo da matriz de correlação Rx
    def calc_matriz_correlacao(self):
        self.matriz_correlacao_x = np.corrcoef(self.matriz_observações)
        return self.matriz_correlacao_x

    # Calculo da matriz de correlação no espaço normal padrão (Rz)
    def matriz_correlacao_nataf(self) -> np.ndarray:
        distribuicoes_uqpy = [mapeamento_uqpy(va) for va in self.vetor_va_cust]

        nataf_obj = Nataf(distributions=distribuicoes_uqpy, corr_x=self.
 ↪matriz_correlacao_x)

        Rz = nataf_obj.corr_z # Matriz de corelação zij
        self.matriz_correlacao_z = Rz
```

```
        return Rz

    # Matriz de eliminação da correlação via decomposição de Cholesky e calculo␣
↪dos Jacobianos Z -> Y
    def decomposicao_cholesky(self) -> np.ndarray:
        B = np.linalg.cholesky(self.matriz_correlacao_z)
        B_inv = np.linalg.inv(B)

        L = np.linalg.inv(B.T)
        Jyz = np.linalg.inv(L) # Jacobiano Jyz
        Jzy = L # Jacobiano Jzy

        return Jyz, Jzy

    # Matriz de eliminação da correlação via decomposição de decomposição␣
↪ortogonal e calculo dos Jacobianos Z -> Y
    def descorrelacao_autovetores(self) -> np.ndarray:
        Rz = self.matriz_correlacao_z

        # W é o vetor de autovalores
        # A_barra é a matriz onde cada coluna é um autovetor de Rz
        W, A_barra = np.linalg.eigh(Rz)

        # Construção da diagonal da matriz inversa dos auto-valores
        Lambda_inv_sqrt = np.diag(1.0 / np.sqrt(W))

        # Matriz de descorrelação A
        A = A_barra @ Lambda_inv_sqrt

        # Jacobiano Jyz
        Jyz = A.T

        # Jacobiano Jzy
        Jzy = np.linalg.inv(A.T)

        return Jyz, Jzy
```

Para verificação do código implementado, considera-se um exemplo composto por 4 variáveis aleatórias com com diferentes distribuições, das quais é conhecido a média (μ) e o desvio padrão (σ):

$$X_1 - N(10, 2)X_2 - LN(12, 3)X_3 - EVI_{mx}(8, 1.5)X_4 - EVI_{min}(5, 1)$$

A matriz de correlação entre as variáveis é:

$$R_x = \begin{pmatrix} 1 & 0.4 & 0 & 0 \\ 0.4 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.5 \\ 0 & 0.5 & 0.5 & 1 \end{pmatrix}$$

[14]:
```
import numpy as np

# Para cada variável é concido os momentos (média e desvio padrão) e a matriz␣
 ↪de correlação Rx

X1 = variavel_aleatoria(distribuicao='normal', nome='X1_Normal', simbolo='X1-N')
X1.conjunto_parametros(10.0, 2.0)

X2 = variavel_aleatoria(distribuicao='lognormal', nome='X2_LogNormal',␣
 ↪simbolo='X2-LN')
X2.calculo_parametros(12.0, 3.0)

X3 = variavel_aleatoria(distribuicao='gumbel_max', nome='X3_GumbelMax',␣
 ↪simbolo='X3-EV1')
X3.calculo_parametros(8.0, 1.5)

X4 = variavel_aleatoria(distribuicao='gumbel_min', nome='X4_GumbelMin',␣
 ↪simbolo='X4-EV1')
X4.calculo_parametros(5.0, 1.0)

vetor_va = [X1, X2, X3, X4]

# Matriz de correlação suposta
Rx_entrada = np.array([
    [1.0, 0.4, 0.0, 0.0],
    [0.4, 1.0, 0.0, 0.5],
    [0.0, 0.0, 1.0, 0.5],
    [0.0, 0.5, 0.5, 1.0]
])

# Matriz de observações, neste caso é necessária apenas para inicial o algoritmo
# uma vez que Rx foi dado, porém deve conter as mesmas dimensões de Rx
matriz_dummy_obs = np.zeros((4, 4))

# Aqui chamamos a classe criada para calcular a tranformação Z -> Y
objeto = vetores_variavel_aleatoria(matriz_dummy_obs, vetor_va)

# Calculo da dimensão do vetor de variáveis aleatórias

dimensão_vetor = objeto.dimensao()
```

```
# Forçamos Rx para a matriz de entrada, eliminando a necessidade de da matriz␣
 ↪de observações
objeto.matriz_correlacao_x = Rx_entrada

# Cálculo da matriz de correlação equivalente
Rz = objeto.matriz_correlacao_nataf()

# Calculo dos Jacobianos Jyz e Jzy via decomposição de Cholesky
J_yz_cho, J_zy_cho = objeto.decomposicao_cholesky()

# Calculo dos Jacobianos Jyz e Jzy via decomposição ortogonal
J_yz_ort, J_zy_ort = objeto.decomposicao_cholesky()

teste = J_zy_ort @ J_yz_ort


# Impressão dos resultados

print(f"Dimensão do vetor de variáveis aleatórias: {dimensão_vetor}") #␣
 ↪Dimensão do vetor de variáveis aleatrorias
print(f"Vetor de variáveis aleatórias: {[va.simbolo for va in vetor_va]}") #␣
 ↪Vetor de variáveis aleatórias
print(f"Matriz de correlação Rx: \n{Rx_entrada}") # Matriz de correlação Rx
print(f"Matriz de correlação equivalente Rz (Modelo de Nataf): \n{Rz}") #␣
 ↪Matriz de correlação equivalente Rz
print(f"Jacobianos de eliminação de correlação segundo deocmposição de␣
 ↪Cholesky")
print(f"Jyz = \n{J_yz_cho}")
print(f"Jyz = \n{J_zy_cho}")
print(f"Jacobianos de eliminação de correlação segundo deocmposição ortogonal")
print(f"Jyz = \n{J_yz_ort}")
print(f"Jyz = \n{J_zy_ort}")
```

```
Dimensão do vetor de variáveis aleatórias: (4,)
Vetor de variáveis aleatórias: ['X1-N', 'X2-LN', 'X3-EV1', 'X4-EV1']
Matriz de correlação Rx:
[[1.  0.4 0.  0. ]
 [0.4 1.  0.  0.5]
 [0.  0.  1.  0.5]
 [0.  0.5 0.5 1. ]]
Matriz de correlação equivalente Rz (Modelo de Nataf):
[[ 1.00000000e+00  4.06139733e-01 -2.48801311e-16  2.14724891e-16]
 [ 4.06139733e-01  1.00000000e+00 -8.61369527e-16  5.12223591e-01]
 [-2.05270737e-16 -9.06034205e-16  1.00000000e+00  5.15427932e-01]
 [ 1.82422364e-16  5.12223591e-01  5.15427932e-01  1.00000000e+00]]
Jacobianos de eliminação de correlação segundo deocmposição de Cholesky
Jyz =
```

```
[[ 1.00000000e+00  4.06139733e-01 -2.05270737e-16  1.38777878e-16]
 [ 0.00000000e+00  9.13810986e-01 -9.00257948e-16  5.60535602e-01]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00  5.15427932e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  6.48177356e-01]]
Jyz =
[[ 1.00000000e+00 -4.44446105e-01 -1.94845401e-16  3.84351385e-01]
 [ 0.00000000e+00  1.09431821e+00  9.85168664e-16 -9.46352583e-01]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00 -7.95195832e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.54278762e+00]]
Jacobianos de eliminação de correlação segundo deocmposição ortogonal
Jyz =
[[ 1.00000000e+00  4.06139733e-01 -2.05270737e-16  1.38777878e-16]
 [ 0.00000000e+00  9.13810986e-01 -9.00257948e-16  5.60535602e-01]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00  5.15427932e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  6.48177356e-01]]
Jyz =
[[ 1.00000000e+00 -4.44446105e-01 -1.94845401e-16  3.84351385e-01]
 [ 0.00000000e+00  1.09431821e+00  9.85168664e-16 -9.46352583e-01]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00 -7.95195832e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.54278762e+00]]
```