DOCUMENTAÇÃO ESPAÇO GEOMÉTRICO NEGADO

Introdução

O intuito desta documentação é deixar claro o processo de desenvolvimento do código, com as classes e funções explicadas, além de uma descrição de como o projeto funciona.

Como executar o projeto

Instalando as dependência

Para rodar o projeto, primeiro, é necessário instalar as bibliotecas necessárias. Para isso, siga os seguintes passos:

1. Clone o repositório para sua máquina local.
2. Crie e ative um ambiente virtual (opcional, mas recomendado):

```
python –m venv env
source env/bin/activate  # Para Linux/macOS
.\env\Scripts\activate   # Para Windows
```

3. Instale as dependências com o comando:

```
pip install –r requirements.txt
```

Executando o projeto

1. Após instalar as dependências, execute o código com o comando:

```
 python setup.py
```

   Resultando na seguinte saida:

```
Arquivos disponíveis:
1. antenas_SE.csv
2. antenas_DF.csv
0. Usar configuração padrão
Selecione o número do arquivo desejado (ou 0 para padrão):
```

2. Recebemos três opções após executar o `setup.py`: a primeira é `antenas_SE.csv`, a segunda é `antenas_DF.csv`, ou ainda a configuração padrão. Mais adiante, falaremos sobre os arquivos CSV. Para este caso, utilizaremos a opção zero.

```
Usar configuração padrão.
Arquivos disponíveis:
1. DF_Heights.tif
2. SE_Heights.tif.
0. Usar configuração padrão
Selecione o número do arquivo desejado (ou 0 para padrão):
```

3. Após selecionar o primeiro arquivo, deve-se escolher o arquivo referente ao terreno. Lembre-se de que devemos sempre escolher os mesmos arquivos. Caso, no passo anterior, tenha sido selecionado `antenas_DF.csv`, agora deve-se escolher `DF_Heights.tif`. Para este caso, utilizaremos a opção zero.

```
484. 1013200486
485. 1013249949
486. 1013200923
487. 1013200737
488. 1014472110
Escolha o índice do número da estação a ser analisada (ou 0 para pa
```

4. Após selecionar o arquivo referente ao terreno, uma lista com todas as antenas disponíveis será exibida. Essa lista é proveniente do arquivo CSV. Selecione uma antena qualquer ou a opção zero como padrão. Para este exemplo, utilizaremos a opção padrão.

```
Estação selecionada: 698607260
Torre interferente 1
Torre interferente 2
Torre interferente 3
Torre interferente 4
Torre interferente 5
Torre interferente 6
Torre interferente 7
Torre interferente 8
Torre interferente 9
SINR: -28.26 dB, SNR: -28.09 dB, Sinal: -115.09 dBm, Interferencia:
```

5. Agora, é iniciado o processo de cálculo, no qual serão registrados todos os pontos com SINR menor ou igual a zero. Também recebemos um log com as informações sobre o número de antenas interferentes e os dados da antena em si.

```
KML salvo com sucesso!!!
Mapa salvo com sucesso!!!
Dados salvo com sucesso!!!
```

6. No final da execução, é informado que nossos dados foram salvos em um arquivo CSV na pasta `dados/mosaico/saida` e que um mapa foi salvo com nosso polígono na pasta `mapas/saida`.

[image]

Estrutura do Projeto

O projeto foi estruturado em módulos, sendo que cada um possui uma funcionalidade específica. Há um módulo dedicado ao cálculo do SINR, um módulo responsável por cálculos simples, como a conversão de unidades (por exemplo, de MHz para Hz), e, por fim, um módulo focado em cálculos relacionados a mapas, como altitudes do terreno, distâncias entre dois pontos geográficos e a criação de um mapa com um polígono.

```
projeto_SINR/
│
├── config.ini                          # Arquivo de configuração
```

```
├── README.md                          # Documentação do projeto
├── dados/                             # Diretório de dados
│   ├── antenas/                       # Dados de antenas
│   │   ├── mosaico/                   # Dados de antenas mosaico
│   │   │   ├── antenas_DF.csv         # Dados de antenas para DF
│   │   │   └── antenas_SE.csv         # Dados de antenas para SE
│   │   └── saida/                     # Saída de processamento de antenas
│   ├── img/                           # Imagens
│   ├── kml/                           # Arquivos kml
│   └── mapas/                         # Dados de mapas
│       ├── altitude/                  # Arquivos de altitude
│       │   ├── DF_Heights.tif         # Altitude do terreno DF
│       │   └── SE_Heights.tif         # Altitude do terreno SE
│       └── saida/                     # Saída de processamento de mapas g
├── docs/                              # Arquivos de documentação de guia
│   ├── documentacao.md                # Documentação completa do projeto
│   ├── documentacao.pdf               # Documentação completa do projeto
│   └── guia_de_uso.md                 # Guia de uso rápido do projeto
├── env/                               # Ambiente virtual
├── requirements.txt                   # Dependências do projeto
├── setup.py                           # Arquivo de inicialização do proje
└── src/                               # Código-fonte do projeto
    ├── main.py                        # Arquivo principal do projeto
    ├── mapa/                          # Módulo para cálculos e criação de
    │   ├── __init__.py                # Inicialização do módulo mapa
    │   ├── Calcular_Altitude.py       # Cálculos de altitude
    │   ├── Calculos_Mapa.py           # Funções de cálculos relacionados
    │   └── Criar_Mapa.py              # Criação de mapas
    ├── poligono/                      # Módulo para criação de polígonos
    │   ├── __init__.py                # Inicialização do módulo poligono
    │   └── gerar_poligono.py          # Arquivo com a função para a criaç
    ├── sinr/                          # Módulo para cálculos de SINR
    │   ├── __init__.py                # Inicialização do módulo sinr
    │   └── Calcular_SINR.py           # Funções do cálculo de SINR
    └── util/                          # Módulo utilitário
        ├── __init__.py                # Inicialização do módulo util
        └── converter.py               # Funções de conversão
```

Arquivos de saída

O objetivo deste código é encontrar o espaço geométrico negado, calculando o SINR das antenas de telecomunicação. Para isso, são utilizados dados como faixa de frequência, potência, largura de faixa, entre outros. Como saída, o código gera três arquivos: HTML, KML e CSV.

1. **HTML**:

- O arquivo HTML exibe um mapa com todas as antenas no município selecionado.
- Também são exibidos os setores da antena analisada, assim como os dados de SINR com os aparelhos móveis na borda da célula.

1. **KML**:

- O arquivo KML gerado pode ser utilizado pelo usuário em diversos mapas, como no Google Earth, permitindo a visualização da área de cobertura da antena.

1. **CSV**:

- O arquivo CSV contém todas as informações de cada ponto móvel.
- Seus dados são:
    - **Número da Estação**: Número de identificação da estação.
    - **Área de Cobertura (m^2)**: Área de cobertura da antena.
    - **SINR (dB)**: SINR do ponto móvel.
    - **SNR (dB)**: SNR do ponto móvel.
    - **Sinal (dBm)**: Sinal obtido no ponto móvel.
    - **Interferência (dBm)**: Interferência obtida no ponto móvel.
    - **Ruído (dBm)**: Ruído obtido no ponto móvel.
    - **Distância (m)**: Distância do ponto móvel até a antena.
    - **Azimute da Antena (°)**: Azimute da antena em relação ao norte verdadeiro.
    - **Azimute do Móvel (°)**: Azimute do móvel em relação ao norte verdadeiro.
    - **Elevação da Antena (°)**: Elevação da antena.
    - **Elevação do Móvel (°)**: Elevação do móvel em relação à antena.
    - **EIRP (dBm/Hz)**: Valor da potência isotrópica radiada equivalente.
    - **Perda no Espaço (dB)**: Valor da perda no espaço livre.
    - **Frequência (MHz)**: Faixa de frequência da antena.
    - **Potência (W)**: Potência de transmissão da antena.
    - **Ganho (dBi)**: Ganho da antena.
    - **Largura da Faixa (Hz)**: Largura de faixa da antena.
    - **Altura da Antena (m)**: Altura da antena.
    - **Latitude do Móvel**: Latitude do ponto móvel.
    - **Longitude do Móvel**: Longitude do ponto móvel.
    - **Altitude da Antena (m)**: Altitude da antena.
    - **Altitude do Móvel (m)**: Altitude do móvel.

Códigos do projeto

Arquivo de inicialização

Arquivo `setup.py`:

```
import os
import configparser

def listar_arquivos(caminho, extensao=".csv"):
    #Lista arquivos de uma extensão específica no caminho.
    return [f for f in os.listdir(caminho) if f.endswith(extensao)]

def selecionar_arquivo(arquivos, padrao=None):
    #Permite ao usuário selecionar um arquivo entre os disponíveis.
    print("Arquivos disponíveis:")
    for i, arquivo in enumerate(arquivos, 1):
        print(f"{i}. {arquivo}")
    if padrao:
        print("0. Usar configuração padrão")
```

```
    while True:
        escolha = input("Selecione o número do arquivo desejado (ou 0 p
        if escolha == "0" and padrao:
            print("Usando configuração padrão.")
            return padrao
        elif escolha.isdigit() and 1 <= int(escolha) <= len(arquivos):
            return arquivos[int(escolha) - 1]
        else:
            print("Escolha inválida. Tente novamente.")

def atualizar_config(arquivo_antena, arquivo_altitude, config_path="con
    #Atualiza o arquivo de configuração com os arquivos escolhidos.
    config = configparser.ConfigParser()
    config.read(config_path)
    config['DEFAULT']['arquivo_antenas'] = arquivo_antena
    config['DEFAULT']['arquivo_altitude'] = arquivo_altitude
    with open(config_path, 'w') as configfile:
        config.write(configfile)
    print(f"Arquivo de configuração atualizado: {arquivo_antena}, {arqu

if __name__ == "__main__":
    # Carrega o arquivo de configuração
    config_path = 'config.ini'
    config = configparser.ConfigParser()
    config.read(config_path)

    # Obtém os caminhos das pastas e arquivos padrão
    antenas_folder = config['PATHS']['antenas_folder']
    altitudes_folder = config['PATHS']['altitudes_folder']
    padrao_antena = config['DEFAULT']['arquivo_antenas']
    padrao_altitude = config['DEFAULT']['arquivo_altitude']

    # Lista e seleciona arquivo de antenas
    arquivos_antenas = listar_arquivos(antenas_folder, ".csv")
    if not arquivos_antenas:
        print(f"Nenhum arquivo encontrado na pasta {antenas_folder}.")
        arquivo_escolhido_antena = padrao_antena
        print("Usando configuração padrão para antenas.")
    elif len(arquivos_antenas) == 1:
        arquivo_escolhido_antena = arquivos_antenas[0]
        print(f"Apenas um arquivo de antena encontrado: {arquivo_escolh
    else:
        arquivo_escolhido_antena = selecionar_arquivo(arquivos_antenas,

    # Lista e seleciona arquivo de altitudes
    arquivos_altitude = listar_arquivos(altitudes_folder, ".tif")
    if not arquivos_altitude:
        print(f"Nenhum arquivo encontrado na pasta {altitudes_folder}."
        arquivo_escolhido_altitude = padrao_altitude
        print("Usando configuração padrão para altitudes.")
    elif len(arquivos_altitude) == 1:
        arquivo_escolhido_altitude = arquivos_altitude[0]
```

```
    print(f"Apenas um arquivo de altitude encontrado: {arquivo_esco
else:
    arquivo_escolhido_altitude = selecionar_arquivo(arquivos_altitu

# Atualiza o arquivo de configuração
atualizar_config(
    os.path.join(antenas_folder, arquivo_escolhido_antena),
    os.path.join(altitudes_folder, arquivo_escolhido_altitude),
    config_path=config_path
)

# Direciona para o main.py
os.system("python src/main.py")
```

1. **Importações**

- `os`: Fornece funções para interagir com o sistema operacional, como listar arquivos e executar comandos no terminal.
- `configparser`: Permite ler, modificar e salvar arquivos de configuração (.ini), que armazenam parâmetros do programa.

1. **Funções**

- **Função** `listar_arquivos(caminho, extensao=".csv")`:
    - A função `listar_arquivos` busca todos os arquivos com uma extensão específica dentro de um diretório e retorna uma lista com seus nomes.

  Parâmetros:

    - `caminho` (str): O caminho do diretório onde os arquivos serão buscados.
    - `extensao` (str, opcional): A extensão dos arquivos a serem listados (padrão: `.csv`).

  Retorno:

    - `lista` (list): Uma lista contendo os nomes dos arquivos encontrados com a extensão especificada.

- **Função** `selecionar_arquivo(arquivos, padrao=None)`:
    - A função `selecionar_arquivo` exibe uma lista de arquivos disponíveis e permite que o usuário escolha um deles digitando seu número correspondente. Caso uma configuração padrão seja fornecida, o usuário pode optar por usá-la.

  Parâmetros:

    - `arquivos` (list): Lista de arquivos disponíveis para seleção.
    - `padrao` (str, opcional): Nome do arquivo padrão que pode ser escolhido.

  Retorno:

    - `arquivo_selecionado` (str): O nome do arquivo escolhido pelo usuário ou o arquivo padrão.

- **Função** `atualizar_config(arquivo_antena, arquivo_altitude, config_path="config.ini")`:
    - A função `atualizar_config` modifica um arquivo de configuração (`config.ini`), atualizando as referências aos arquivos de antenas e altitude escolhidos.

  Parâmetros:

    - `arquivo_antena` (str): O nome do arquivo de antenas a ser salvo na configuração.
    - `arquivo_altitude` (str): O nome do arquivo de altitude a ser salvo na configuração.
    - `config_path` (str, opcional): O caminho do arquivo de configuração (padrão: "`config.ini`").

  Retorno:

    - Nenhum retorno explícito, mas atualiza o arquivo de configuração e exibe uma mensagem de confirmação.

1. **Execução Principal (`__main__`)**

```
if __name__ == "__main__":
```

- Garante que o código só será executado se o arquivo for rodado diretamente, e não importado como módulo.

1. **Carrega a Configuração**

```
config_path = 'config.ini'
config = configparser.ConfigParser()
config.read(config_path)
```

- Lê o arquivo `config.ini`, que contém informações sobre os diretórios e arquivos padrão.

1. **Obtém Caminhos e Arquivos Padrão**

```
antenas_folder = config['PATHS']['antenas_folder']
altitudes_folder = config['PATHS']['altitudes_folder']
padrao_antena = config['DEFAULT']['arquivo_antenas']
padrao_altitude = config['DEFAULT']['arquivo_altitude']
```

- Obtém os diretórios onde estão os arquivos de antenas e altitudes.
- Recupera os arquivos padrão definidos anteriormente.

1. **Lista e Seleciona Arquivo de Antenas**

```
arquivos_antenas = listar_arquivos(antenas_folder, ".csv")
if not arquivos_antenas:
    print(f"Nenhum arquivo encontrado na pasta {antenas_folder}.")
    arquivo_escolhido_antena = padrao_antena
    print("Usando configuração padrão para antenas.")
elif len(arquivos_antenas) == 1:
    arquivo_escolhido_antena = arquivos_antenas[0]
```

```
    print(f"Apenas um arquivo de antena encontrado: {arquivo_escolh
else:
    arquivo_escolhido_antena = selecionar_arquivo(arquivos_antenas,
```

- Lista os arquivos `.csv` na pasta de antenas.
- Se não houver arquivos, usa o arquivo padrão.
- Se houver apenas um, seleciona automaticamente.
- Se houver mais de um, pede para o usuário escolher.

1. **Lista e Seleciona Arquivo de Altitudes**

```
arquivos_altitude = listar_arquivos(altitudes_folder, ".tif")
if not arquivos_altitude:
    print(f"Nenhum arquivo encontrado na pasta {altitudes_folder}."
    arquivo_escolhido_altitude = padrao_altitude
    print("Usando configuração padrão para altitudes.")
elif len(arquivos_altitude) == 1:
    arquivo_escolhido_altitude = arquivos_altitude[0]
    print(f"Apenas um arquivo de altitude encontrado: {arquivo_esco
else:
    arquivo_escolhido_altitude = selecionar_arquivo(arquivos_altitu
```

- Mesmo processo da parte anterior, mas para arquivos de altitudes (`.tif`).

1. **Atualiza a Configuração**

```
atualizar_config(
    os.path.join(antenas_folder, arquivo_escolhido_antena),
    os.path.join(altitudes_folder, arquivo_escolhido_altitude),
    config_path=config_path
)
```

- Atualiza o arquivo `config.ini` com os arquivos escolhidos.

1. Executa o `main.py`

```
os.system("python src/main.py")
```

- Executa o arquivo `main.py`, que provavelmente faz o processamento dos dados com os arquivos selecionados.

Arquivo principal do projeto

Arquivo `main.py`:

```
from sinr.Calcular_SINR import Calcular_SINR
from mapa.Criar_Mapa import Criar_Mapa
from poligono.gerar_poligono import gerar_poligono
from folium import Map
import pandas as pd
import configparser
import os
```

```
def carregar_antenas(caminho_arquivo):
    # Carrega os dados do arquivo CSV.
    try:
        antenas = pd.read_csv(caminho_arquivo)
        df = pd.DataFrame(antenas)
        return df
    except FileNotFoundError:
        print(f"Erro: O arquivo {caminho_arquivo} não foi encontrado.")
        return None


if __name__ == '__main__':
    # Lê o arquivo de configuração
    config = configparser.ConfigParser()
    config.read('config.ini')

    # Obtém os valores do arquivo de configuração
    arquivo_antenas = config['PATHS']['arquivo_antenas']
    print(arquivo_antenas)

    # Carrega os dados do arquivo CSV
    df = carregar_antenas(arquivo_antenas)

    if df is not None:
        # Obtém os valores únicos de NumEstacao
        numeros_estacoes = df['NumEstacao'].unique()
        print("Números de estação disponíveis:")
        for i, num in enumerate(numeros_estacoes, start=1):
            print(f"{i}. {num}")

        # Solicita ao usuário que escolha o índice da estação
        while True:
            try:
                indice_escolhido = int(input("Escolha o índice do númer
                if indice_escolhido == 0:
                    antena_analisada = int(config['DEFAULT']['antena_an
                    break
                elif 1 <= indice_escolhido <= len(numeros_estacoes):
                    antena_analisada = numeros_estacoes[indice_escolhid
                    break
                else:
                    print("Índice inválido. Tente novamente.")
            except ValueError:
                print("Entrada inválida. Por favor, insira um número vá

        print(f"Estação selecionada: {antena_analisada}")

        # Filtra os dados para a antena escolhida
        dados_filtrados = df[df['NumEstacao'] == antena_analisada][['Az

        # Lat e Lon de referência para o mapa
        LATITUDE = float(df['Latitude'][0])
        LONGITUDE = float(df['Longitude'][0])
```

```
# Cria uma lista de tuplas de Azimute e AnguloElevacao
chaves = list(zip(dados_filtrados['Azimute'], dados_filtrados['

# Remove duplicatas
chaves_unicas = list(set(chaves))

# Ordena a lista pelo Azimute do menor ao maior
chaves_ordenadas = sorted(chaves_unicas, key=lambda x: x[0])

df_result = []

# Loop para calcular o SINR para cada azimute
for az, el in chaves_ordenadas:
    distancia = 5
    az_ini = 0
    valor = True
    while valor:
        # Cria a instância da classe Calcular_SINR para cada pa
        calcular_sinr = Calcular_SINR(antena_analisada, az, el,

        # Chama o método de cálculo de SINR
        resultado = calcular_sinr.calculo_SINR(distancia, az_in

        # Acesso ao valor de SINR a partir do dicionário
        sinr_value = resultado['SINR']

        # Condição para aumentar a distância caso o SINR seja m
        if sinr_value <= 0:
            df_result.append(resultado)
            distancia = 5
            az_ini += 10
            # Verifica se todos os ângulos foram calculados
            if az_ini >= 360:
                valor = False
                break
            else:
                continue
        else:
            distancia += 5

df_result = pd.DataFrame(df_result)

area_poli = gerar_poligono(antena_analisada, df_result)
print('KML salvo com sucesso!!!')

df_result['Area_Coberta'] = area_poli

df_result = df_result[['Area_Coberta'] + [col for col in df_res

df_result['NumEstacao'] = antena_analisada

df_result = df_result[['NumEstacao'] + [col for col in df_resul
```

```
# Cria o mapa
mapa = Map(location=[LATITUDE, LONGITUDE], zoom_start=17, max_z

mapa_process = Criar_Mapa(mapa, df, df_result)
mapa_result = mapa_process.processar_resultados()

nome_arquivo_saida_mapa = f'mapa_SINR_{antena_analisada}.html'

# Caminho para salvar o mapa
mapa_nome = os.path.join("dados", "mapas", "saida", nome_arquiv

# Salva o mapa
mapa.save(mapa_nome)
print('Mapa salvo com sucesso!!!')

# Renomeando as colunas
df_result = df_result.rename(columns={
    'NumEstacao': 'Número da Estação',
    'Area_Coberta': 'Área de Cobertura (m²)',
    'SINR': 'SINR (dB)',
    'SNR': 'SNR (dB)',
    'Sinal': 'Sinal (dBm)',
    'Interferencia': 'Interferência (dBm)',
    'Ruido': 'Ruído (dBm)',
    'Distancia': 'Distância (m)',
    'Azimute_Antena': 'Azimute da Antena (°)',
    'Azimute_Movel': 'Azimute do Móvel (°)',
    'Elevacao_Antena': 'Elevacao da Antena (°)',
    'Elevacao_Movel': 'Elevacao do Móvel (°)',
    'EIRP': 'EIRP (dBm/Hz)',
    'Free_Space': 'Perda no Espaço (dB)',
    'Frequencia': 'Frequência (MHz)',
    'Potencia': 'Potencia (W)',
    'Ganho': 'Ganho (dBi)',
    'Largura_Faixa': 'Largura da Faixa (Hz)',
    'Altura': 'Altura da Antena (m)',
    'Latitude': 'Latitude do Móvel',
    'Longitude': 'Longitude do Móvel',
    'Altitude_Antena': 'Altitude da Antena (m)',
    'Altitude_Movel': 'Altitude do Móvel (m)'
})

# Define o nome do arquivo de saída com base em antena_analisad
nome_arquivo_base = os.path.basename(arquivo_antenas)
sufixo_nome = nome_arquivo_base.split('_')[-1]
nome_arquivo_saida = f"{antena_analisada}_{sufixo_nome}"

# Caminho completo para salvar o arquivo
caminho_saida = os.path.join('dados/antenas/saida/', nome_arqui

# Garante que o diretório de saída existe
os.makedirs('dados/antenas/saida/', exist_ok=True)
```

```
# Salva o DataFrame como CSV
df_result.to_csv(caminho_saida, index=False)

print("Dados salvo com sucesso!!!")
```

1. **Importações**

- Essas bibliotecas e módulos são usados para diferentes funcionalidades do código:
  - `Calcular_SINR` e `Criar_Mapa`: Classes personalizadas (provavelmente desenvolvidas por você) que lidam com cálculos de SINR e criação de mapas.
  - `folium.Map`: Biblioteca para criar e visualizar mapas interativos.
  - `pandas`: Usada para manipulação de dados tabulares em forma de DataFrame.
  - `configparser`: Lê configurações de arquivos .ini.
  - `os`: Manipula caminhos e arquivos no sistema.
  - `simplekml`: Gera arquivos KML para exibição em ferramentas como Google Earth.
  - `scipy.spatial.ConvexHull`: Calcula o fecho convexo, útil para criar áreas em torno de pontos.
  - `shapely.geometry` e `shapely.ops`: Manipulam formas geométricas, como polígonos.
  - `pyproj.Transformer`: Converte coordenadas entre diferentes sistemas de referência, como latitude/longitude e UTM.

1. Funções

- **Função** `carregar_antenas`
  - Objetivo: Lê um arquivo CSV e retorna um DataFrame.
  - Detalhes:
    - Tenta carregar os dados do arquivo especificado.
    - Caso o arquivo não exista, retorna None e exibe um erro.

1. Código Principal `(if __name__ == '__main__')` 3.1 Carregar Configurações

- Lê um arquivo config.ini para obter caminhos de arquivos e valores padrão, como:
  - O caminho do arquivo CSV com dados de antenas.
  - Qual antena será analisada por padrão.

3.2 Processamento de Dados

- **Carregar Dados**: Usa `carregar_antenas` para ler o arquivo CSV.
- **Selecionar Antena**:
  - Lista todas as antenas (`NumEstacao`) disponíveis.
  - Solicita ao usuário que escolha qual analisar ou usa o valor padrão do arquivo de configuração.
- **Filtrar Dados**: Seleciona apenas os dados da antena escolhida.

3.3 Calcular SINR

- Para cada combinação de azimute e elevação:

- Cria um objeto da classe `Calcular_SINR`.
- Calcula o SINR para diferentes distâncias e ângulos iniciais (`az_ini`), ajustando até cobrir todos os ângulos.
- Armazena os resultados em uma lista.

3.4 Criar Polígono

- Usa os resultados do cálculo de SINR para criar polígonos (com `gerar_poligono`).
- Calcula a área total coberta pelas antenas.

3.5 Criar Mapa

- Usa o `folium` para criar um mapa centrado na latitude e longitude da antena analisada.
- Processa os resultados do cálculo de SINR com a classe `Criar_Mapa` para exibir a cobertura da antena.
- Salva o mapa em formato HTML para visualização posterior.

3.6 Salvar Resultados

- Salva os resultados do cálculo e a área coberta em um arquivo CSV no diretório especificado.

20. **Explicação Final**

- O código principal é responsável por carregar as configurações iniciais, processar os dados das antenas e calcular o **SINR (Signal-to-Interference-plus-Noise Ratio)** para diferentes combinações de azimute e ângulo de elevação.
- Após o cálculo do SINR, o código gera **polígonos** que representam a área de cobertura da antena e salva os resultados em um arquivo CSV. Além disso, o código cria um **mapa interativo** que ilustra visualmente a cobertura da antena, facilitando a análise da distribuição do sinal.

Classe Cálculo do SINR:

O principal objetivo deste projeto é calcular o SINR ao redor das antenas de telecomunicações, com o intuito de encontrar a área negada ou espaço geométrico negado. Para solucionar esse problema, foi desenvolvida a classe `Calcular_SINR`, que está no arquivo `Calcular_SINR.py`.

Método construtor da classe

```
class Calcular_SINR:
    def __init__(self, antena_analisada, azimute, elevacao, dados):
        self.df = dados
        self.id = antena_analisada
        self.az = azimute
        self.el = elevacao
        indice = self.df[self.df['NumEstacao'] == self.id].index

        # Se o índice não estiver vazio, obtenha os valores corresponde
        if not indice.empty:
```

```
            self.freq = dados['FreqTxMHz'].iloc[indice[0]]
            self.ganho = dados['GanhoAntena'].iloc[indice[0]]
            self.pot = dados['PotenciaTransmissorWatts'].iloc[indice[0]
            self.alt = dados['AlturaAntena'].iloc[indice[0]]
            self.ang_meia = dados['AnguloMeiaPotenciaAntena'].iloc[indi
            self.designacao_faixa = dados['DesignacaoEmissao'].iloc[ind
            self.lat = dados['Latitude'].iloc[indice[0]]
            self.lon = dados['Longitude'].iloc[indice[0]]
        else:
            # Lida com o caso em que não encontra um índice corresponde
            raise ValueError(f"Estação {antena_analisada} não encontrad

        self.altura_movel = 1.5  # m

        # Constantes
        self.pi = pi
        self.c = 299_792_458 # m/s

        # Classe com calculos geograficos
        self.calcular_mapa_antena = Calculos_Mapa(self.lat, self.lon, s
```

O método construtor `__init__` da classe `Calcular_SINR` tem como objetivo inicializar os atributos de uma instância dessa classe com base em parâmetros fornecidos e dados presentes em um DataFrame. Vamos analisar o que está acontecendo, passo a passo:

Parâmetros:

- antena_analisada: O ID da antena a ser analisada. Esse valor é usado para filtrar os dados no DataFrame.
- azimute (ou az): O azimute da antena, em graus. Representa a direção de transmissão da antena.
- elevacao (ou el): A elevação da antena, relacionada ao ângulo da antena em relação ao solo.
- dados: Um DataFrame contendo informações sobre as antenas e suas características.

Ações do método:

1. **Inicialização dos atributos**:`self.df`: Atribui o DataFrame dados ao atributo df, que é um DataFrame contendo as informações sobre as antenas.`self.id`: Atribui o valor de antena_analisada ao atributo id, que é o identificador da antena.`self.az`: Atribui o valor de azimute ao atributo az, o azimute da antena.`self.el`: Atribui o valor de elevacao ao atributo el, a elevação da antena.

2. **Busca no DataFrame**:A linha `indice = self.df[self.df['NumEstacao'] == self.id].index` procura no DataFrame o índice da linha onde a coluna `NumEstacao` corresponde ao valor de `self.id`. Esse índice é utilizado para localizar os dados específicos da antena no DataFrame.

3. **Verificação se a antena foi encontrada**:Se o índice não estiver vazio `if not indice.empty`, o código obtém os valores das colunas correspondentes à antena, como:`FreqTxMHz` (frequência de transmissão em MHz),`GanhoAntena` (ganho da antena),`PotenciaTransmissorWatts` (potência do transmissor em watts),`AlturaAntena` (altura da antena),`AnguloMeiaPotenciaAntena` (ângulo de meia potência da antena),`DesignacaoEmissao` (designação da emissão),`Latitude` e

`Longitude` (coordenadas geográficas da antena).Esses valores são atribuídos aos atributos da classe, como `self.freq`, `self.ganho`, `self.pot`, etc.

4. Caso a antena não seja encontrada:Se o índice estiver vazio, significa que a antena com o ID fornecido não foi encontrada no DataFrame. Nesse caso, o código levanta um erro (ValueError), informando que a estação não foi encontrada.

5. Inicialização de atributos adicionais:

   `self.alt_movel`: Define a altura do objeto móvel (altura de uma pessoa que está sendo considerado para cálculos) como 1,5 metros.

   `self.pi`: Atribui o valor de pi.

   `self.c`: Define a constante da velocidade da luz em metros por segundo (299.792.458 m/s).

6. Instanciação de outro objeto: `self.calcular_mapa_antena`: Cria uma instância da classe `Calculos_Mapa` utilizando as coordenadas geográficas da antena `self.lat`, `self.lon`, o azimute da antena `self.az` e a altura da antena `self.alt`.

Função para calcular o ganho gaussiano

```
# Calculo do ganho gaussiano
def calcular_ganho(self, ganho, az_0, el_0, az, el, ang_meia):
    '''
        Para o calculo foi normalizado os ângulos com o valor de 0° tan
        para o azimute quanto para a elevação, fazendo um melhor uso
        do modelo gaussiano
    '''
    if az_0 == 0 and el_0 == 0:
        if az > 180:
            az -= 360
        resultado = ganho - 12 * (((el - el_0) / 7)**2 + ((az - az_0) /
        return resultado
    else:
        if az_0 == 0:
            el -= el_0
            el_0 = 0
            if az > 180:
                az -= 360
            resultado = ganho - 12 * (((el - el_0) / 7)**2 + ((az - az_
            return resultado
        elif el_0 == 0:
            az -= az_0
            az_0 = 0
            if az > 180:
                az -= 360
            elif az < -180:
                az += 360
            resultado = ganho - 12 * (((el - el_0) / 7)**2 + ((az - az_
            return resultado
        else:
            az -= az_0
```

```
        el -= el_0
        az_0 = 0
        el_0 = 0
        if az > 180:
            az -= 360
        elif az < -180:
            az += 360
        resultado = ganho - 12 * (((el - el_0) / 7)**2 + ((az - az_
        return resultado
```

A função `calcular_ganho` calcula o ganho gaussiano de uma antena com base na diferença angular entre os valores de azimute e elevação do ponto de referência e o ponto alvo. O modelo utilizado normaliza os ângulos de azimute e elevação em relação a 0°, permitindo um uso mais eficiente da fórmula gaussiana.

Parâmetros:

- `ganho` (float): O ganho máximo da antena, em dBi.
- `az_0` (float): O azimute de referência, em graus.
- `el_0` (float): A elevação de referência, em graus.
- `az` (float): O azimute do ponto alvo, em graus.
- `el` (float): A elevação do ponto alvo, em graus.
- `ang_meia` (float): O ângulo de meia potência (half-power beamwidth) da antena, em graus.

Retorno:

- resultado (float): O ganho da antena em direção ao ponto alvo, em dB.

Função para calcular potência isotrópica radiada equivalente (EIRP)

```
# Calcula a potência isotrópica radiada equivalente
def calcular_eirp(self, ganho, pot):
    return eirp_to_hz(watts_to_dBm(pot) + ganho)
```

A função `calcular_eirp` calcula a **Potência Isotrópica Radiada Equivalente** (EIRP) em uma frequência específica, utilizando como entrada o ganho da antena e a potência de transmissão. A EIRP representa a potência efetiva transmitida na direção de maior ganho de uma antena isotrópica equivalente, considerando o ganho da antena e a potência de entrada.

Parâmetros:

- `ganho` (float): O ganho da antena em decibéis isotrópicos (dBi).
- `pot` (float): A potência transmitida em watts.

Retorno:

- resultado (float): O EIRP da antena e retornada em dBm/Hz.

Função para calcular a perda no espaço livre

```
# Calcula a perda no espaço livre
def calcular_perda_no_espaco(self, distancia):
    return 20 * log10((4 * self.pi * mhz_to_hz(self.freq) * distancia)/
```

A função `calcular_perda_no_espaco` calcula a perda de propagação no espaço livre (em dB), considerando a frequência do sinal e a distância até o receptor.

A fórmula é baseada no modelo de perda em espaço livre, que depende do logaritmo da razão entre o produto da frequência (em Hz) e a distância pela velocidade da luz.

Parâmetros:

- `distancia` (float): Distância entre a antena transmissora e o receptor, em metros.
- `self.freq` (float): Frequência do sinal, em MHz, convertida para Hz pela função `mhz_to_hz`.
- `self.c` (float): Velocidade da luz no vácuo, em m/s.
- `self.pi` (float): O valor de π (pi).

Retorno:

- perda (float): A perda no espaço livre, em dB.

Função para calcular o ruído

```
# Calcula o ruido da antena
def calcular_ruido(self):
    return −174 + 10 * log10(largura_canal_hz(self.designacao_faixa)) +
```

A função calcular_ruido calcula o ruído total da antena em dBm, considerando a largura de faixa utilizada.

O cálculo baseia-se no ruído em um canal específico, somando o ruído de referência de -174 dBm/Hz, o fator logarítmico da largura de banda e uma constante de 7 dB, que representa a figura de ruido.

Figura de ruído e fator de ruído são medidas de degradação da relação sinal-ruído (SNR), causada por componentes em uma cadeia de sinal de radiofrequência. É um número pelo qual o desempenho de um amplificador ou receptor de rádio pode ser especificado, com valores mais baixos indicando melhor desempenho.

Parâmetros:

- `self.lar_faixa` (float): A largura de faixa da antena, em Hz.
- `largura_canal_hz` (função): Retorna a largura de banda em Hz, usada para o cálculo.

Retorno:

- ruído (float): O valor do ruído em dBm.

Função para calcular a interferência

```
# Calcula a interferencia sofrida no determinado ponto
def calculo_interferencia(self, lat_movel, lon_movel, alt_movel):
    contador_i = 0
    s_i = 0
    for i, row in self.df.iterrows():
        # estaçao ou azimute diferente
        if ((row['Azimute'] != self.az) or ((row['Azimute'] == self.az)
```

```
            latitude_i = row['Latitude']
            longitude_i = row['Longitude']
            potencia_i = row['PotenciaTransmissorWatts']
            ganho_i = row['GanhoAntena']
            altura_i = row['AlturaAntena']
            azimute_i = row['Azimute']
            elevacao_i = row['AnguloElevacao']
            meia_pot_i = row['AnguloMeiaPotenciaAntena']

            mapa_antena_i = Calculos_Mapa(latitude_i, longitude_i, azim
            # Calcula a distancia do móvel para a antena
            distancia_para_movel = mapa_antena_i.calcular_distanca_Am(l

            if distancia_para_movel <= 1000:#and (azimute_direcao_movel
                # Calcula a altitude da antena
                altitude_antena_i = altitudeCalculator.calc_altitude(la
                #altitude_movel = altitudeCalculator.calc_altitude(lat_
                # Calcula a distancia do móvel até o ponto mais alto da
                distancia_diagonal_movel = mapa_antena_i.calc_hipotenus
                # Calcula o azimute do móvel em relação a antena interf
                azimute_direcao_movel = mapa_antena_i.calcular_diferenc
                # Calcula a elevação do móvel em relação a antena inter
                elevacao_movel_i = mapa_antena_i.calcular_elevacao(dist
                # Calcula o ganho gaussiano dBm
                ganho_gaussiano = self.calcular_ganho(ganho_i, azimute_
                # Calcula o EIRP dBm/Hz
                eirp_antena = self.calcular_eirp(ganho_gaussiano, poten
                # Calcula a perda do espaço livre dB
                espaco_livre = self.calcular_perda_no_espaco(distancia_
                # Calcula o Sinal dBm
                sinal = eirp_antena - espaco_livre
                s_i += dBm_to_mW(sinal)
                print(f'Torre interferente {contador_i + 1}')
                contador_i += 1
            else:
                continue
    # Verificando se realmente existe torres interferentes
    if s_i == 0:
        return 0
    else:
        return mW_to_dBm(s_i)
```

A função `calculo_interferencia` calcula o nível de interferência (em dBm) sofrido em um ponto móvel (definido por latitude, longitude e altitude) causado por torres interferentes próximas.

O cálculo considera várias características de cada torre, como frequência, potência, ganho da antena, azimute, elevação, e aplica modelos de propagação e geometria para determinar o impacto no ponto móvel.

Parâmetros:

- `lat_movel` (float): Latitude do ponto móvel.
- `lon_movel` (float): Longitude do ponto móvel.

- `alt_movel` (float): Altitude do ponto móvel, em metros.

Retorno:

- Interferência (float): O nível de interferência total no ponto móvel, em dBm. Retorna 0 se nenhuma torre interferente for encontrada.

Detalhes importantes:

- **Iteração nas torres**: Verifica torres interferentes que atendem às condições (frequência, largura de faixa e distância).
- **Cálculos geométricos**: Determina a distância, azimute e elevação do ponto móvel em relação às torres interferentes.
- **Cálculo do sinal**: Calcula o sinal recebido (dBm) combinando EIRP, ganho gaussiano e perdas no espaço livre.
- **Soma da interferência**: Converte o sinal para mW, acumula os valores e retorna o total em dBm.

Função para calcular o SINR

```
def calculo_SINR(self, distancia, azimute):
    # Calcula a altitude da antena
    altitude_antena = altitudeCalculator.calc_altitude(self.lat, self.l
    # Calcula a lat e lon móvel
    lat_movel, lon_movel = self.calcular_mapa_antena.calcular_lat_lon(a
    # Calcula a altitude do móvel
    altitude_movel = altitudeCalculator.calc_altitude(lat_movel, lon_mo
    # Calcula a distancia do móvel até o ponto mais alto da antena
    distancia_diagonal = self.calcular_mapa_antena.calc_hipotenusa((sel
    # Calcula a elevação do móvel em relação a antena
    elevacao_movel = self.calcular_mapa_antena.calcular_elevacao(distan

    # Corrigindo margem de erro do modelo gaussiano
    if self.el == 0:
        if elevacao_movel < -8 or distancia < self.alt:
            elevacao_movel_calc = self.el
        else:
            elevacao_movel_calc = elevacao_movel
    else:
        if self.el > 0:
            elevacao_movel_dif = elevacao_movel - self.el
        else:
            elevacao_movel_dif = elevacao_movel + self.el

        if elevacao_movel_dif < -8 or distancia < self.alt:
            elevacao_movel_calc = self.el
        else:
            elevacao_movel_calc = elevacao_movel

    # Calcula o ganho gaussiano dBm
    ganho_gaussiano = self.calcular_ganho(self.ganho, self.az, self.el,
    # Calcula o EIRP dBm/Hz
    eirp_antena = self.calcular_eirp(ganho_gaussiano, self.pot)
    # Calcula a perda do espaço livre dB
```

```
    espaco_livre = self.calcular_perda_no_espaco(distancia)
    # Calcula o Ruido dBm
    ruido = self.calcular_ruido()
    # Calcula o Sinal dBm
    sinal = eirp_antena - espaco_livre
    # Calcula o SNR dB
    snr = sinal - ruido
    # Calcula a interferencia dBm
    inter = self.calculo_interferencia(lat_movel, lon_movel, altitude_m
    # Calcula o SINR dB
    if inter == 0: # Casos em que a Antena esta em um ponto isolado sem
        inter = nan # Retorna Nan para esses casos já que realmente não
        sinr = sinal - ruido
    else:
        sinr = sinal - 10 * log10(dBm_to_mW(inter) + dBm_to_mW(ruido))

    # Depuração do código
    print(f'SINR: {sinr:.2f} dB, SNR: {snr:.2f} dB, Sinal: {sinal:.2f}

    # dados que serão retornados
    resultado = {
        'SINR': sinr,
        'SNR': snr,
        'Sinal': sinal,
        'Interferencia': inter,
        'Ruido': ruido,
        'Distancia': distancia,
        'Azimute_Antena': self.az,
        'Azimute_Movel': azimute,
        'Elevacao_Antena': self.el,
        'Elevacao_Movel': elevacao_movel,
        'EIRP': eirp_antena,
        'Free_Space': espaco_livre,
        'Frequencia': self.freq,
        'Potencia': self.pot,
        'Ganho': ganho_gaussiano,
        'Largura_Faixa': largura_canal_hz(self.designacao_faixa),
        'Altura': self.alt,
        'Latitude': lat_movel,
        'Longitude': lon_movel,
        'Altitude_Antena': altitude_antena,
        'Altitude_Movel': altitude_movel
    }

    return resultado
```

A função `calculo_SINR` calcula o SINR (relação sinal para interferência mais ruído) em um ponto móvel, levando em consideração a interferência de outras torres e o ruído.

A função realiza cálculos geométricos, como a determinação de distâncias, elevação e azimute do ponto móvel em relação à antena. Também ajusta o ganho da antena considerando os efeitos do ambiente. Com esses dados, calcula o Sinal (EIRP menos a perda no espaço livre), o SNR (relação sinal-ruído) e o SINR (considerando interferências).

Parâmetros:

- `distancia` (float): A distância entre a antena e o ponto móvel, em metros.
- `azimute` (float): O azimute do ponto móvel em relação à antena, em graus.

Retorno:

- resultado (dict): Um dicionário contendo os cálculos de SINR, SNR, sinal, interferência, ruído, distâncias, azimute, elevação, EIRP, e outros dados relacionados.

Classe Criar_Mapas

O objetivo da classe `Criar_Mapa` e criar um mapa com todas as antenas fornecidas pelos arquivos na pasta `\mosaico`, a classe se encontra no arquivo que está no arquivo `Criar_Mapa.py`.

Método construtor da classe

```
class Criar_Mapa():
    """
    Classe para criação e manipulação de um mapa interativo com antenas
    pontos móveis e polígonos, utilizando a biblioteca Folium. Permite
    adicionar marcadores, desenhar polígonos e linhas com base em dados
    de antenas e SINR.
    """
    def __init__(self, mapa, dados_antena, dados_sinr):
        self.df_antena = dados_antena
        self.df_sinr = dados_sinr
        self.mp = mapa
        self.azimute_cores = {}  # Dicionário para armazenar azimutes e
        self.cores_disponiveis = deque(['red', 'green', 'blue', 'orange

        # Caminho para o ícone
        self.icon_1 = 'dados/img/antena.png'
        self.icon_2 = 'dados/img/celular.png'
```

O método construtor `__init__` da classe `Criar_Mapa` tem como objetivo inicializar os atributos de uma instância dessa classe com base nos parâmetros fornecidos e nas variáveis internas necessárias para gerar o mapa.

Parâmetros:

- `mapa`: Representa o objeto do mapa onde as antenas e outros dados serão exibidos.
- `dados_antena`: Um DataFrame contendo as informações das antenas que serão usadas para a construção do mapa.
- `dados_sinr`: Um DataFrame contendo as informações sobre o SINR (Signal-to-Interference-plus-Noise Ratio), que pode ser utilizado para realizar cálculos no mapa.

Ações do método:

1. **Inicialização dos atributos**:

- `self.df_antena`: Atribui o DataFrame `dados_antena` ao atributo `df_antena`, que armazenará as informações sobre as antenas.

- `self.df_sinr`: Atribui o DataFrame `dados_sinr` ao atributo `df_sinr`, que contém os dados sobre o SINR.
- `self.mp`: Atribui o valor de mapa ao atributo mp, que representa o mapa onde as antenas serão visualizadas e interagidas.

1. **Inicialização de coleções**:

- `self.azimute_cores`: Inicializa um dicionário vazio, azimute_cores, que será usado para associar azimutes de antenas a cores, permitindo que diferentes direções (azimutes) possam ser visualmente destacadas no mapa.
- `self.cores_disponiveis`: Inicializa uma fila (deque) com cores disponíveis, que serão usadas para atribuir diferentes cores a cada azimute de antena. A fila é circular, permitindo que as cores sejam reutilizadas conforme necessário.

1. **Definição de caminhos para ícones**:

- `self.icon_1`: Atribui o caminho para a imagem de um ícone de antena, utilizado no mapa para representar as antenas.
- `self.icon_2`: Atribui o caminho para a imagem de um ícone de celular, que será utilizado para representar a posição de um ponto móvel no mapa.

Função get_color

```
def get_color(self, azimute):
    # Se o azimute já tiver uma cor atribuída, retorna a cor
    if azimute in self.azimute_cores:
        return self.azimute_cores[azimute]

    # Caso contrário, atribui a próxima cor disponível da fila
    cor = self.cores_disponiveis[0]
    # Muda para a próxima cor apenas se a cor já foi usada pelo azimute
    while cor in self.azimute_cores.values():
        self.cores_disponiveis.rotate(-1)  # Move a cor usada para o fi
        cor = self.cores_disponiveis[0]

    # Atribui a cor ao azimute atual e retorna
    self.azimute_cores[azimute] = cor
    return cor
```

A função `get_color` atribui uma cor a um azimute, garantindo que cada azimute tenha uma cor única e que as cores disponíveis sejam reutilizadas de maneira eficiente, sem repetição entre azimutes. A função utiliza uma fila circular de cores e, se necessário, ajusta a cor até encontrar uma disponível para o azimute solicitado.

Parâmetros:

- `azimute` (float): O azimute para o qual a cor será atribuída. O azimute representa a direção de um ponto em relação ao norte e será utilizado como chave para armazenar a cor correspondente.

Ações do método:

1. **Verificação da cor atribuída ao azimute**:

- Se o azimute já tiver uma cor associada no dicionário `self.azimute_cores`, a cor

existente é retornada imediatamente.

1. **Seleção de uma cor disponível**:

- Caso o azimute ainda não tenha uma cor atribuída, a função seleciona a primeira cor disponível na fila `self.cores_disponiveis`.

1. **Ajuste de cor caso já tenha sido usada**:

- Se a cor selecionada já foi atribuída a outro azimute, a fila de cores é rotacionada (movida para a esquerda) até que uma cor não utilizada seja encontrada.

1. **Atribuição e retorno da cor**:

- A cor é associada ao azimute no dicionário `self.azimute_cores` e retornada como o resultado da função.

Retorno:

- cor (str): A cor atribuída ao azimute, que será uma das cores da fila `self.cores_disponiveis` (como 'red', 'green', 'blue', etc.).

Função add_polygon_to_map

```
def add_polygon_to_map(self, dados, color):
    # Se houver dados, cria o polígono
    if dados:
        polygon_geojson = geojson.Feature(
            geometry=geojson.Polygon([
                [(p['Longitude'], p['Latitude']) for p in dados]
            ])
        )

        folium.GeoJson(
            polygon_geojson,
            style={
                'color': color,
                'fillColor': color,
                'fillOpacity': 0.5,
                'weight': 2
            }
        ).add_to(self.mp)
```

A função `add_polygon_to_map` adiciona um polígono representando uma área delimitada por pontos geográficos ao mapa, utilizando a biblioteca `folium` para a visualização. A função aceita dados contendo coordenadas geográficas e aplica um estilo ao polígono, incluindo cor, opacidade e espessura das bordas.

Parâmetros:

- `dados` (list): Uma lista de dicionários, onde cada dicionário contém as chaves `Longitude` e `Latitude` que representam as coordenadas dos vértices do polígono.
- `color` (str): Uma string representando a cor a ser usada no polígono. Essa cor é aplicada tanto ao preenchimento quanto às bordas do polígono.

Ações do método:

1. **Verificação da presença de dados**:

- A função verifica se o parâmetro dados contém elementos. Caso contrário, o polígono não será criado.

1. **Criação do GeoJSON do polígono**:

- Utiliza a biblioteca `geojson` para criar uma estrutura GeoJSON que define o polígono.
- As coordenadas para o polígono são extraídas do parâmetro dados e dispostas no formato apropriado para um geojson.Polygon.

1. Adição do polígono ao mapa:

- A função utiliza `folium.GeoJson` para renderizar o polígono no mapa, aplicando os seguintes estilos:
  - Cor da borda (color): Define a cor das bordas do polígono.
  - Cor de preenchimento (fillColor): Define a cor do interior do polígono.
  - Opacidade de preenchimento (fillOpacity): Define a transparência do preenchimento.
  - Espessura da borda (weight): Define a largura das bordas do polígono.
- O polígono estilizado é adicionado ao mapa referenciado por `self.mp`.

Retorno:

- A função não possui retorno explícito, mas adiciona o polígono estilizado ao objeto de mapa `self.mp`.

Função adicionar_antenas

```
def adicionar_antenas(self):
    # Dicionário para armazenar os azimutes por estação
    azimutes_por_estacao = {}

    # Primeiro, itere sobre o DataFrame para agrupar os azimutes por es
    for i, row in self.df_antena.iterrows():
        num_estacao = row['NumEstacao']

        # Se a estação já existir no dicionário, adicione o azimute à l
        if num_estacao in azimutes_por_estacao:
            azimutes_por_estacao[num_estacao].append(f"{row['Azimute']}
        else:
            # Caso contrário, inicialize uma nova lista com o primeiro
            azimutes_por_estacao[num_estacao] = [f"{row['Azimute']}°"]

    # Agora itere novamente para adicionar os marcadores ao mapa com os
    coordenadas_adicionadas = set()  # Para evitar duplicatas de coorde

    for i, row in self.df_antena.iterrows():
        coordenada_atual = (row['Latitude'], row['Longitude'])
        num_estacao = row['NumEstacao']
```

```
        # Pega todos os azimutes associados à estação atual e concatena
        saida_azimutes = ", ".join(azimutes_por_estacao[num_estacao])

        # Verificar se a coordenada já foi adicionada
        if coordenada_atual not in coordenadas_adicionadas:
            if row['FreqTxMHz'] == self.df_sinr['Frequencia'][0]:
                popup_content = (f"""
                <b>Nome da Estação:</b> {row['NomeEntidade']}<br>
                <b>Latitude:</b> {row['Latitude']}<br>
                <b>Longitude:</b> {row['Longitude']}<br>
                <b>Designação:</b> {row['DesignacaoEmissao']}<br>
                <b>Número da Torre:</b> {row['NumEstacao']}<br>
                <b>Frequência:</b> {row['FreqTxMHz']} MHz<br>
                <b>Potência do Transmissor:</b> {row['PotenciaTransmiss
                <b>Ganho da Antena:</b> {row['GanhoAntena']} dBi<br>
                <b>Altura da Antena:</b> {row['AlturaAntena']} m<br>
                <b>Azimute:</b> {saida_azimutes}<br>
                <b>Ângulo de Elevação:</b> {row['AnguloElevacao']}°<br>
                <b>Ângulo de Meia Potência:</b> {row['AnguloMeiaPotenci
                """)

                # Adiciona o marcador no mapa com os azimutes da estaçã
                folium.Marker(
                    location=[row['Latitude'], row['Longitude']],
                    popup=folium.Popup(popup_content, max_width=300),
                    tooltip=popup_content,
                    icon=CustomIcon(self.icon_1, (50, 50))  # Ícone per
                ).add_to(self.mp)

                # Adicionar a coordenada ao conjunto para evitar duplic
                coordenadas_adicionadas.add(coordenada_atual)
```

A função `adicionar_antenas` adiciona marcadores no mapa para representar antenas com informações detalhadas e agrupadas por estação. A função utiliza um dicionário para organizar azimutes associados a cada estação e evita duplicação de marcadores para coordenadas iguais.

Parâmetros:

- `self.df_antena` (DataFrame): Contém informações sobre as antenas, como coordenadas, azimutes, frequência, potência, e outros atributos técnicos.
- `self.df_sinr` (DataFrame): Utilizado para verificar a frequência de interesse.
- `self.icon_1` (str): Caminho para o ícone personalizado dos marcadores.
- `self.mp`: Objeto de mapa criado com a biblioteca folium, onde os marcadores serão adicionados.

Ações do método:

1. **Agrupamento de azimutes por estação**:

- Cria um dicionário `azimutes_por_estacao` onde a chave é o número da estação (`NumEstacao`) e o valor é uma lista contendo os azimutes associados a essa estação.
- Percorre o DataFrame `self.df_antena` para agrupar os azimutes em listas.
- Cada azimute é formatado como uma string com o símbolo °.

1. **Evitar duplicação de marcadores**:

- Cria um conjunto `coordenadas_adicionadas` para armazenar as coordenadas de marcadores já adicionados. Isso impede que múltiplos marcadores sejam colocados na mesma localização.

1. **Iteração para adicionar marcadores**:

- Itera novamente pelo DataFrame para adicionar os marcadores no mapa.
- Para cada linha:
    - Recupera as coordenadas (`Latitude` e `Longitude`) e verifica se já foram adicionadas.
    - Gera uma string com os azimutes associados à estação atual usando o dicionário `azimutes_por_estacao`.
    - Verifica se a frequência de transmissão da antena corresponde à frequência de interesse (`self.df_sinr['Frequencia'][0]`).
    - Cria um conteúdo de popup com informações detalhadas da antena, como nome da estação, frequência, potência, altura, azimutes, entre outros.

1. Adição do marcador ao mapa:

- Utiliza a biblioteca `folium` para adicionar um marcador na posição da antena com:
    - Popup: Mostra as informações da antena ao clicar no marcador.
    - Tooltip: Mostra informações ao passar o mouse sobre o marcador.
    - Ícone personalizado: Adiciona um ícone com dimensões definidas (`50x50`) para representar a antena.

1. Registro da coordenada:

- Adiciona a coordenada ao conjunto `coordenadas_adicionadas` para evitar duplicação.

Retorno:

- A função não possui retorno explícito, mas modifica o objeto de mapa self.mp, adicionando marcadores para as antenas que atendem aos critérios definidos.

Função adicionar_retas_az_antena

```
def adicionar_retas_az_antena(self):
    # Iterar sobre as linhas do DataFrame `antenas_df`
    for i, row in self.df_antena.iterrows():
        # Calculos para o mapa
        self.calc_mapa = Calculos_Mapa(self.df_sinr['Latitude'][i], sel
        # Filtra `df` para verificar se a estação e a frequência corres
        filtro = (self.df_sinr['NumEstacao'] == row['NumEstacao']) & (s
        if filtro.any():  # Verifica se há alguma linha correspondente
            # Calcula a reta para o azimute da antena
            reta_latitude, reta_longitude = self.calc_mapa.calcular_ret

            # Desenha a linha (reta) apontando para o azimute
            folium.PolyLine(
                locations=[(row['Latitude'], row['Longitude']), (reta_l
```

```
            color="orange",
            weight=2.5
        ).add_to(self.mp)
```

A função `adicionar_retas_az_antena` desenha retas no mapa indicando a direção do ganho máximo de cada antena. Essa funcionalidade é utilizada para fins de visualização, facilitando a compreensão da orientação das antenas com base no seu azimute.

Parâmetros:

- `self.df_antena` (DataFrame): Contém informações detalhadas das antenas, incluindo localização, número da estação, frequência e azimute.
- `self.df_sinr` (DataFrame): Contém informações de referência das antenas, como latitude, longitude, azimute e altura.
- `self.mp`: Objeto do mapa `folium`, onde as retas serão adicionadas.

Ações do método:

1. **Iteração pelo DataFrame** `df_antena`:

- Percorre todas as linhas de `self.df_antena` para processar cada antena individualmente.

1. **Cálculo para a direção do azimute**:

- Instancia um objeto da classe `Calculos_Mapa` para calcular as coordenadas da reta apontando para a direção do azimute.
- Os parâmetros fornecidos à classe são:
    - Latitude e longitude da antena de referência.
    - Azimute da antena.
    - Altura da antena.

1. **Filtro para correspondência**:

- Filtra o DataFrame `self.df_sinr` para identificar linhas onde o número da estação (`NumEstacao`) e a frequência de transmissão (`FreqTxMHz`) correspondem aos valores da antena atual.
- O filtro garante que as retas sejam desenhadas apenas para antenas válidas e correspondentes.

1. **Cálculo das coordenadas da reta**:

- Utiliza o método `calcular_reta` da instância de `Calculos_Mapa` para obter as coordenadas finais da reta (latitude e longitude do ponto na direção do azimute).

1. **Desenho da reta no mapa**:

- Utiliza folium.PolyLine para desenhar a reta entre a posição inicial da antena (latitude, longitude) e o ponto calculado na direção do azimute.
- Configurações visuais da reta:
    - Cor: Laranja, indicando a direção do ganho máximo.
    - Espessura: 2.5, para destacar as linhas no mapa.

1. **Adição ao mapa**:

- Cada reta é adicionada ao objeto `self.mp`, tornando-a visível no mapa interativo.

Retorno:

- A função não retorna valores, mas modifica o objeto de mapa `self.mp`, adicionando retas que indicam visualmente a direção do ganho máximo de cada antena.

Função adicionar_pontos_moveis

```
# Adiciona os celulares com informações para análise
def adicionar_pontos_moveis(self):
    for i, ponto in self.df_sinr.iterrows():
        popup_movel = (f"""
        <b>SINR</b>: {ponto['SINR']:.2f} dB <br>
        <b>Azimute da Antena</b>: {ponto['Azimute_Antena']}° <br>
        <b>Azimute Móvel</b>: {ponto['Azimute_Movel']}° <br>
        <b>Distância</b>: {ponto['Distancia']} m
        """)
        if ponto['Distancia'] > 20:
            folium.Marker(
                location=[ponto['Latitude'], ponto['Longitude']],
                popup=folium.Popup(popup_movel, max_width=300),
                tooltip=popup_movel,
                icon=CustomIcon(self.icon_2, (20, 20))
            ).add_to(self.mp)
        else:
            folium.Marker(
                location=[ponto['Latitude'], ponto['Longitude']],
                popup=folium.Popup(popup_movel, max_width=300),
                tooltip=popup_movel,
                icon=CustomIcon(self.icon_2, (10, 10))
            ).add_to(self.mp)
```

A função `adicionar_pontos_moveis` adiciona marcadores representando dispositivos móveis (celulares) em um mapa interativo, com informações relevantes para análise de sinal. Essa função utiliza dados de um DataFrame para determinar a localização, o SINR, o azimute e a distância de cada ponto móvel em relação a uma antena.

Parâmetros:

- `self.df_sinr` (DataFrame): Contém informações dos dispositivos móveis, incluindo:
    - **SINR** (Signal-to-Interference-plus-Noise Ratio): Relação sinal-ruído em dB.
    - **Azimute_Antena**: Direção da antena em graus.
    - **Azimute_Movel**: Direção do ponto móvel em graus.
    - **Distancia**: Distância entre o ponto móvel e a antena em metros.
    - **Latitude** e **Longitude**: Localização geográfica do ponto móvel.
- `self.mp`: Objeto de mapa folium, onde os marcadores serão adicionados.
- `self.icon_2`: Ícone personalizado utilizado para representar os pontos móveis no mapa.

Ações do método:

1. **Iteração pelo DataFrame** `df_sinr`:

- Percorre todas as linhas de `self.df_sinr`, processando cada ponto móvel individualmente.

1. **Criação do conteúdo do popup**:

- Para cada ponto móvel, cria uma descrição detalhada com:
    - **SINR**: Mostrado com duas casas decimais.
    - **Azimute da Antena**: Direção da antena em graus.
    - **Azimute Móvel**: Direção do ponto móvel em graus.
    - **Distância**: Distância entre o ponto móvel e a antena em metros.

1. Verificação da distância:

- Se a distância for maior que 20 metros:
    - Adiciona um marcador com um ícone maior (20x20).
- Caso contrário:
    - Adiciona um marcador com um ícone menor (10x10).

1. Adição do marcador ao mapa:

- Cada marcador é posicionado nas coordenadas geográficas do ponto móvel (`Latitude`, `Longitude`).
- O marcador inclui:
    - **Popup**: Mostra as informações detalhadas ao clicar no marcador.
    - **Tooltip**: Exibe as informações ao passar o mouse sobre o marcador.
    - **Ícone personalizado**: Representa visualmente o ponto móvel com tamanhos ajustados pela distância.

Retorno:

- A função não retorna valores, mas modifica o objeto de mapa `self.mp`, adicionando marcadores interativos para cada ponto móvel.

Função processar_azimutes

```
# Função responsavel por adicionar os polígonos e cores no mapa
def processar_azimutes(self):
    dados_por_azimute = {}
    for i, ponto in self.df_sinr.iterrows():
        azimute = ponto['Azimute_Antena']
        if azimute not in dados_por_azimute:
            dados_por_azimute[azimute] = []
        dados_por_azimute[azimute].append(ponto)

    for azimute, dados in dados_por_azimute.items():
        cor = self.get_color(azimute)
        self.add_polygon_to_map(dados, cor)
```

A função `processar_azimutes` organiza dados de pontos móveis agrupados por azimutes das antenas e exibe polígonos no mapa, usando cores distintas para representar cada grupo. Ela utiliza métodos auxiliares para gerar as cores e desenhar os polígonos.

Objetivo:

Adicionar polígonos ao mapa, agrupados e diferenciados pelas cores, baseando-se nos azimutes das antenas associadas aos pontos móveis.

Estrutura e Etapas:

1. Dicionário dados_por_azimute:

- Criação: Inicia um dicionário vazio para armazenar os pontos móveis agrupados pelo azimute.
- Iteração pelo DataFrame `df_sinr`:
    - Para cada ponto móvel no DataFrame:
        - Obtém o valor do azimute associado (`ponto['Azimute_Antena']`).
        - Verifica se o azimute já existe como uma chave no dicionário:
            - Se não existir: Cria uma nova chave para o azimute com uma lista vazia.
            - Se existir: Adiciona o ponto móvel à lista correspondente.

1. Iteração sobre os grupos de azimutes:

- Percorre o dicionário `dados_por_azimute`, onde cada chave é um azimute e o valor associado é a lista de pontos móveis.
- Para cada azimute:
    - Determina a cor:
        - Chama o método `get_color(azimute)` para obter uma cor correspondente ao azimute.
    - Adiciona o polígono ao mapa:
        - Chama o método `add_polygon_to_map(dados, cor)`:
            - `dados`: Lista de pontos móveis associados ao azimute.
            - `cor`: Cor obtida para representar visualmente o azimute no mapa.

Métodos Auxiliares:

1. get_color(azimute):
2. add_polygon_to_map(dados, cor):

Retorno:

- A função não retorna valores, mas modifica o objeto de mapa `self.mp` com polígonos coloridos que indicam a direção (azimute) das antenas em relação aos pontos móveis.

Função processar_resultados

```
def processar_resultados(self):
    self.adicionar_pontos_moveis()
    self.adicionar_antenas()
    self.processar_azimutes()
```

A função `processar_resultados` é um método central que orquestra a execução de três operações principais relacionadas à visualização e organização de dados no mapa. Essa função chama três métodos auxiliares em sequência para garantir que todos os elementos necessários sejam processados e adicionados ao mapa interativo.

Objetivo: Processar e exibir no mapa as informações dos pontos móveis, antenas e agrupamentos por azimutes, consolidando os resultados de análise visual.

Etapas:

1. `self.adicionar_pontos_moveis()`:

- **Propósito**: Adiciona ao mapa os marcadores que representam os pontos móveis.
- **Detalhes**:
    - Cada ponto móvel tem um popup com informações como SINR, azimute do ponto e da antena, e a distância.
    - O tamanho do ícone varia conforme a distância do ponto (maior para distâncias maiores).

1. `self.adicionar_antenas()`:

- **Propósito**: Adiciona ao mapa as antenas representadas como marcadores com informações detalhadas.
- **Detalhes**:
    - As antenas são agrupadas por estação (com base no número da estação) e exibem os azimutes associados a cada estação.
    - Informações adicionais, como frequência, potência e altura, são incluídas no popup.
    - Evita duplicação de marcadores com base nas coordenadas.

1. `self.processar_azimutes()`:

- **Propósito**: Agrupa pontos móveis por azimutes e desenha polígonos coloridos no mapa.
- **Detalhes**:
    - Organiza os pontos móveis por azimutes das antenas.
    - Gera polígonos no mapa, cada um com uma cor distinta para representar visualmente a influência de um azimute específico.

Classe Calcular_Altitude

Método construtor da classe

```
class Calcular_Altitude:
    """
    Classe para calcular a altitude de um ponto geográfico utilizando d
```

```
    A classe utiliza a biblioteca rasterio para manipular e acessar os
    em um arquivo, permitindo calcular a altitude com base em coordenad
    """
    def __init__(self, file_path):
        self.file_path = file_path
        self.src = None
```

O método construtor `__init__` da classe `Calcular_Altitude` é responsável por inicializar os atributos necessários para o processamento de dados de altitude ao criar uma instância da classe.

Parâmetros:

- `file_path`: Representa o caminho do arquivo de entrada. Ele será usado para carregar e processar informações relacionadas à altitude.

Ações do método:

1. **Armazenar o caminho do arquivo**:
    - O parâmetro file_path é atribuído ao atributo self.file_path, permitindo que a classe acesse o caminho do arquivo posteriormente para operações como leitura ou análise de dados.
3. **Inicializar o atributo** `src`:
    - O atributo `self.src` é inicializado com o valor `None`, indicando que nenhuma fonte de dados foi carregada ainda. Isso serve como um espaço reservado que será atualizado em métodos futuros.

Função open_raster

```
# Abre o arquivo raster
def open_raster(self):
    if self.src is None:
        try:
            self.src = rasterio.open(self.file_path)
        except FileNotFoundError:
            raise RuntimeError("ERRO: Arquivo não encontrado")
        except Exception as e:
            raise RuntimeError(f"ERRO inesperado ao abrir o arquivo: {e
```

A função `open_raster` é responsável por abrir um arquivo raster especificado no atributo `file_path` da classe. Ela utiliza a biblioteca `rasterio`, que é amplamente usada para manipulação de dados geoespaciais.

Estrutura e Etapas:

1. **Verificação de** `self.src`:
    - A função começa verificando se o atributo `self.src` é `None`. Se já houver um arquivo raster carregado em `self.src`, o código não tentará abrir o arquivo novamente.
3. **Tentativa de abrir o arquivo**:

- Caso `self.src` seja `None`, a função tenta abrir o arquivo raster usando a biblioteca `rasterio` com o caminho fornecido em `self.file_path`.
- Se o arquivo for aberto com sucesso, ele é armazenado no atributo `self.src`.

6. **Tratamento de erros**:

   - Se o arquivo não for encontrado, um erro `FileNotFoundError` é capturado, e uma mensagem de erro específica é lançada: "ERRO: Arquivo não encontrado".
   - Caso ocorra outro erro inesperado durante a abertura do arquivo, a exceção genérica é capturada e uma mensagem de erro detalhada é gerada, incluindo o erro original.

Função calc_altitude

```
# Calcula a altitude de um ponto geográfico
def calc_altitude(self, lat, lon):
    if self.src is None:
        self.open_raster()

    try:
        # Converte as coordenadas (longitude, latitude) para o sistema
        row, col = self.src.index(lon, lat)

        # Obtém o valor da altitude para as coordenadas
        altitude = self.src.read(1)[row, col]

        return altitude

    except IndexError:
        return "ERRO: Coordenadas fora do alcance do raster"
    except Exception as e:
        return f"ERRO inesperado: {e}"
```

A função `calc_altitude` é responsável por calcular a altitude de um ponto geográfico específico, com base nas coordenadas de latitude e longitude fornecidas. Ela usa um arquivo raster previamente carregado para acessar os dados de elevação.

Objetivo: Calcular a altitude para um ponto geográfico dado, utilizando um arquivo raster de elevação (como um DEM - Digital Elevation Model) que contém as informações de altitude para uma área geográfica.

Estrutura e Etapas:

1. **Verificação do arquivo raster**:

   - A função verifica se o arquivo raster (`self.src`) já foi carregado:

     ○ Se `self.src` for `None`, isso significa que o arquivo raster ainda não foi aberto. Nesse caso, a função chama o método `open_raster()` para abrir o arquivo e carregar os dados.

4. **Conversão de coordenadas**:

   - A função usa o método `self.src.index(lon, lat)` da biblioteca rasterio para converter as coordenadas de longitude e latitude fornecidas (`lon`, `lat`) para

o sistema de coordenadas do raster. Esse método retorna os índices de linha e coluna correspondentes no raster para as coordenadas geográficas fornecidas.

6. **Leitura da altitude**:

    - A função lê a camada de dados do raster usando `self.src.read(1)`, que retorna os valores da primeira banda do raster (geralmente a banda de altitude).
    - O valor de altitude correspondente às coordenadas é acessado com os índices de linha (`row`) e coluna (`col`), e é retornado como o valor de altitude.

9. **Tratamento de exceções**:

    - Exceção `IndexError`: Caso as coordenadas fornecidas estejam fora do alcance do raster (por exemplo, fora dos limites geográficos do arquivo), a função captura a exceção `IndexError` e retorna a mensagem `"ERRO: Coordenadas fora do alcance do raster"`.
    - Exceções genéricas: Para qualquer outro erro inesperado, uma exceção genérica é capturada e a mensagem de erro é retornada, com a descrição do erro original.

Retorno:

- Se o cálculo for bem-sucedido, a função retorna o valor da altitude correspondente às coordenadas fornecidas.
- Caso ocorra um erro (como coordenadas fora do alcance do raster ou erro inesperado), a função retorna uma mensagem de erro apropriada.

Função close_raster

```
# Fecha o arquivo raster
def close_raster(self):
    if self.src is not None:
        self.src.close()
        self.src = None
```

A função `close_raster` é responsável por fechar o arquivo raster que foi aberto, liberando os recursos associados e garantindo que o arquivo não permaneça aberto após o uso.

Estrutura e Etapas:

1. **Verificação do arquivo raster**:

    - A função verifica se o arquivo raster (`self.src`) está aberto, ou seja, se `self.src` não é `None`. Isso é importante para evitar erros ao tentar fechar um arquivo que não foi aberto.

3. **Fechamento do arquivo**:

    - Se o arquivo raster estiver aberto (ou seja, `self.src` não for None), a função chama `self.src.close()` para fechar o arquivo.
    - Isso garante que o arquivo seja liberado corretamente e os recursos do sistema sejam liberados, evitando o uso excessivo de memória ou problemas ao tentar reabrir o arquivo.

6. **Limpeza do objeto**:

- Após fechar o arquivo, a função define `self.src` como `None`. Isso indica que o arquivo foi fechado e a variável não se refere mais a um objeto aberto, tornando a instância pronta para ser reutilizada ou descartada sem referências ao arquivo fechado.

Classe Calculos_Mapa

Método construtor da classe

```
class Calculos_Mapa:
    """
    Classe para realizar cálculos relacionados a mapas, como determinaç
    coordenadas geográficas, azimute, distância e elevação com base em
    """
    def __init__(self, latitude, longitude, azimute, altura):
        self.lat = latitude
        self.lon = longitude
        self.az = azimute
        self.alt = altura

        # Constante
        self.RAIO_TERRA = 6371000
```

O método construtor `__init__` da classe `Calculos_Mapa` inicializa os atributos de uma instância da classe com os dados fornecidos. Ele é responsável por configurar os valores iniciais necessários para realizar os cálculos relacionados a mapas, como coordenadas geográficas, azimute, distância e elevação.

Parâmetros:

- `latitude`: Valor da latitude do ponto geográfico de referência, utilizado para cálculos de distância e outros cálculos geográficos.
- `longitude`: Valor da longitude do ponto geográfico de referência, também utilizado em cálculos geográficos.
- `azimute`: O azimute (em graus) relativo ao ponto de referência, que indica a direção a partir do ponto inicial para realizar os cálculos de direção.
- `altura`: A altura do ponto geográfico, possivelmente usada para calcular a elevação ou ajustar cálculos de distância/altitude com base na elevação do terreno.

Ações realizadas pelo método construtor:

1. **Atribuição de valores**:
    - `self.lat` = latitude: Atribui o valor da latitude passada para o atributo `lat` da classe.
    - `self.lon` = longitude: Atribui o valor da longitude passada para o atributo `lon` da classe.
    - `self.az` = azimute: Atribui o valor do azimute passado para o atributo `az` da classe.
    - `self.alt` = altura: Atribui o valor da altura passada para o atributo `alt` da classe.

6. **Definição de constante**:

- `self.RAIO_TERRA = 6371000`: Define a constante `RAIO_TERRA`, que é o raio médio da Terra em metros. Este valor será utilizado em cálculos relacionados a distâncias geográficas e pode ser essencial em cálculos de distâncias em um sistema esférico, considerando a forma da Terra.

Função calc_hipotenusa

```
def calc_hipotenusa(self, altura, distancia):
    return sqrt(altura**2 + distancia**2)
```

A função `calc_hipotenusa` calcula a hipotenusa de um triângulo retângulo, usando o teorema de Pitágoras. Ela recebe dois parâmetros: `altura` e `distancia`, e retorna o valor da hipotenusa.

Parâmetros:

- `altura`: Um dos catetos do triângulo retângulo, geralmente representando a diferença de altura entre dois pontos.
- `distancia`: O outro cateto do triângulo retângulo, que pode representar a distância horizontal entre dois pontos.

Etapas do cálculo:

1. Elevando os catetos ao quadrado: A função eleva `altura` e `distancia` ao quadrado (`altura**2` e `distancia**2`).
2. Somando os quadrados: A soma dos quadrados dos catetos é calculada (`altura**2 + distancia**2`).
3. Calculando a raiz quadrada: A função utiliza a função `sqrt` da biblioteca `math` para calcular a raiz quadrada da soma dos quadrados, resultando no valor da hipotenusa.

Retorno:

- A função retorna um único valor, que é a hipotenusa do triângulo retângulo calculada.

Função calcular_elevacao

```
def calcular_elevacao(self, distancia_horizontal, distancia_vertical):
    angulo = acos(distancia_horizontal / distancia_vertical)
    return -(degrees(angulo))
```

A função `calcular_elevacao` calcula o ângulo de elevação entre dois pontos com base em suas distâncias horizontais e verticais, e retorna o ângulo em graus, com o sinal negativo. Esse tipo de cálculo é utilizado para encontrar o ângulos de inclinação da torre.

Parâmetros:

- `distancia_horizontal`: A distância entre os dois pontos ao longo da linha horizontal (por exemplo, a distância horizontal entre uma torre e um ponto de observação).
- `distancia_vertical`: A distância vertical entre os dois pontos (por exemplo, a diferença de altura entre o ponto de observação e o topo de uma torre).

Etapas do cálculo:

1. Cálculo do arco cosseno: A função `acos()` calcula o arco cosseno (ou inverso do

cosseno) da razão entre a `distancia_horizontal` e a `distancia_vertical`. Essa razão representa o cosseno do ângulo de elevação.
2. Conversão para graus: O valor do ângulo é convertido de radianos para graus utilizando a função `degrees()`.
3. Aplicação do sinal negativo: O valor do ângulo é multiplicado por -1, invertendo o sinal. Isso é feito para representar o ângulo de elevação como um valor negativo, possivelmente indicando uma inclinação para baixo.

Retorno:

- A função retorna o valor do ângulo de elevação em graus, com sinal negativo.

Função calcular_lat_lon

```
# Função para calcular a latitude e longitude do móvel
def calcular_lat_lon(self, azimute, distancia):
    # Converter latitude e longitude para radianos
    lat = radians(self.lat)
    lon = radians(self.lon)

    # Converter azimute para radianos
    azimute = radians(azimute)

    # Calcular a nova latitude
    lat_1 = asin(sin(lat) * cos(distancia / self.RAIO_TERRA) +
                 cos(lat) * sin(distancia / self.RAIO_TERRA) * cos(azimu

    # Calcular a nova longitude
    lon_1 = lon + atan2(sin(azimute) * sin(distancia / self.RAIO_TERRA)
                        cos(distancia / self.RAIO_TERRA) - sin(lat) * s

    # Converter de volta para graus
    lat_graus = degrees(lat_1)
    lon_graus = degrees(lon_1)

    return lat_graus, lon_graus
```

A função `calcular_lat_lon` é responsável por calcular a nova posição geográfica (latitude e longitude) de um ponto móvel, dado um azimute e uma distância a partir de um ponto de referência (torre de antena).

Parâmetros:

- `azimute`: O ângulo de direção em relação ao norte, medido em graus a partir do norte verdadeiro (geralmente entre 0° e 360°).
- `distancia`: A distância entre o ponto de origem (latitude e longitude) e o ponto móvel, geralmente medida em metros.

Etapas do cálculo:

1. Entrada: Latitude, longitude, azimute e distância.
2. Conversão de valores para radianos.
3. Cálculo da nova latitude usando a fórmula trigonométrica esférica.
4. Cálculo da nova longitude usando a fórmula trigonométrica esférica.

5. Conversão dos resultados de volta para graus.
6. Retorno das novas coordenadas.

Retorno:

- A função retorna as novas coordenadas geográficas, `lat_graus` e `lon_graus`, que representam a latitude e a longitude do ponto móvel após mover-se na direção do azimute especificado a partir da posição inicial.

Função calcular_distanca_Am

```
# Função para calcular a distancia da antena para móvel
def calcular_distanca_Am(self, lat2, lon2):
# Converter as latitudes e longitudes de graus para radianos
    lat1 = radians(self.lat)
    lon1 = radians(self.lon)
    lat2 = radians(lat2)
    lon2 = radians(lon2)

    # Diferenças entre as coordenadas
    delta_lat = lat2 - lat1
    delta_lon = lon2 - lon1

    # Aplicar a fórmula de Haversine
    a = sin(delta_lat / 2)**2 + cos(lat1) * cos(lat2) * sin(delta_lon /
    c = 2 * atan2(sqrt(a), sqrt(1 - a))

    # Distância em metros
    antena_movel = self.RAIO_TERRA * c

    return antena_movel
```

A função `calcular_distanca_Am` calcula a distância geodésica (em metros) entre dois pontos na superfície da Terra, utilizando as coordenadas geográficas (latitude e longitude) de ambos os pontos.

Objetivo: Calcular a distância em linha reta (curva esférica) entre dois pontos na Terra, dados por suas coordenadas geográficas (latitude e longitude).

Parâmetros:

- `lat2`: Latitude do segundo ponto (ponto móvel ou destino) em graus.
- `lon2`: Longitude do segundo ponto (ponto móvel ou destino) em graus.

Etapas do cálculo:

1. Entrada: Latitude e longitude de dois pontos (o ponto da antena e o ponto móvel).
2. Conversão de valores para radianos:

    - As latitudes e longitudes dos pontos de origem e destino são convertidas de graus para radianos.

4. Cálculo da diferença entre as coordenadas:

    - Calcula-se a diferença entre as latitudes e longitudes dos dois pontos.

6. Aplicação da fórmula de Haversine:

    - Calcula-se a variável `a` com base nas diferenças de latitude e longitude.
    - Calcula-se o valor de `c` (ângulo central entre os dois pontos) usando a função `atan2`.

9. Cálculo da distância em metros:

    - Multiplica-se o valor de `c` pelo raio da Terra (6371000 metros) para calcular a distância real entre os dois pontos.

11. Retorno: A distância entre os pontos em metros é retornada.

Retorno:

- A função retorna a distância em metros entre o ponto de origem (a antena) e o ponto de destino (o ponto móvel), utilizando a fórmula de Haversine para calcular a distância esférica.

Função calcular_azimute

```
# Gera o azimute correspondente ao móvel
def calcular_azimute(self, lat, lon):
    d_lon = radians(lon - self.lon)
    lat_antena = radians(self.lat)
    lat_movel = radians(lat)

    x = sin(d_lon) * cos(lat_movel)
    y = cos(lat_antena) * sin(lat_movel) - sin(lat_antena) * cos(lat_mo
    azimute = atan2(x, y)

    return degrees(azimute)
```

A função `calcular_azimute` calcula o azimute, ou seja, o ângulo entre o norte geográfico e a linha que conecta uma antena a um ponto móvel, medido no plano horizontal.

Objetivo: Determinar o azimute (ângulo) entre dois pontos geográficos, dados pelas suas latitudes e longitudes.

Parâmetros:

- `lat`: Latitude do ponto móvel em graus.
- `lon`: Longitude do ponto móvel em graus.

Etapas do cálculo:

1. Entrada: Latitude e longitude do ponto móvel e do ponto de referência (antena).
2. Conversão de valores para radianos:

    - Calcula a diferença de longitude (`d_lon`) em radianos.
    - Converte as latitudes do ponto de referência e do ponto móvel para radianos.

5. Cálculo das variáveis auxiliares: `x`: Calculado como o produto do seno de `d_lon` pelo cosseno da latitude do ponto móvel. `y`: Calculado como a diferença entre:

- O produto do cosseno da latitude do ponto de referência pelo seno da latitude do ponto móvel.
- O produto do seno da latitude do ponto de referência, cosseno da latitude do ponto móvel e cosseno de `d_lon`.

8. Cálculo do azimute:

    - Utiliza a função `atan2` para calcular o ângulo em radianos a partir das variáveis `x` e `y`.

10. Conversão para graus:

    - Converte o resultado do azimute de radianos para graus.

12. Retorno: Retorna o azimute em graus.

Retorno:

- O azimute entre a antena e o ponto móvel, medido em graus.

Função calcular_diferenca_azimute

```
def calcular_diferenca_azimute(self, az):
    # Calcula as duas possibilidades de diferença
    result1 = az - self.az
    result2 = (az + 360) - self.az

    # Verifica qual resultado está mais próximo de zero
    if abs(result1) < abs(result2):
        return result1
    else:
        return result2
```

A função `calcular_diferenca_azimute` calcula a diferença entre um azimute fornecido (`az`) e o azimute de referência (`self.az`), retornando o menor ângulo absoluto entre eles. Isso é útil para determinar o deslocamento angular mais curto em uma escala circular (0° a 360°).

Objetivo: Calcular a menor diferença angular entre dois azimutes, considerando que os ângulos podem "circular" ao passar de 360° para 0°.

Parâmetros:

- `az`: Azimute em graus (ângulo fornecido para comparação).

Etapas do cálculo:

1. Entrada: Azimute de referência (`self.az`) e azimute fornecido (`az`).
2. Cálculo das duas possibilidades de diferença angular:

    - `result1`: Diferença direta entre o azimute fornecido e o de referência (`az - self.az`).
    - `result2`: Diferença considerando o efeito de "circularidade" dos ângulos (`(az + 360) - self.az`).

5. Determinação do menor ângulo absoluto:

    - Compara os valores absolutos de `result1` e `result2`.
    - Seleciona a diferença angular com menor magnitude.

8. Retorno: Retorna o menor ângulo, preservando o sinal (positivo ou negativo) para indicar a direção do deslocamento.

Retorno:

- A menor diferença angular entre os dois azimutes, em graus, com sinal indicando a direção.

Função calcular_reta

```
# Gera uma reta na direção do azimute
def calcular_reta(self, distancia_metros=100):
    R = 6378137  #Raio da Terra em metros
    distancia_rad = distancia_metros / R

    azimute_rad = radians(self.az)
    lat_rad = radians(self.lat)
    lon_rad = radians(self.lon)

    nova_lat = asin(sin(lat_rad) * cos(distancia_rad) +
                    cos(lat_rad) * sin(distancia_rad) * cos(azimute
    nova_lon = lon_rad + atan2(sin(azimute_rad) * sin(distancia_rad) *
                               cos(distancia_rad) - sin(lat_rad) *

    return degrees(nova_lat), degrees(nova_lon)
```

A função `calcular_reta` calcula o ponto geográfico final (latitude e longitude) ao percorrer uma reta a partir de um ponto inicial, na direção de um azimute e a uma distância especificada. Ela utiliza fórmulas de navegação esférica para determinar as novas coordenadas.

Objetivo: Determinar o ponto final de uma reta traçada a partir de um ponto inicial, seguindo um azimute e deslocando-se uma distância definida.

Parâmetros:

- `distancia_metros`: Distância a ser percorrida a partir do ponto inicial, em metros (valor padrão é 100 metros).

Etapas do cálculo:

1. Entrada: Latitude, longitude e azimute do ponto inicial, além da distância a ser percorrida.
2. Conversão para radianos:

    - Latitude (`lat`), longitude (`lon`) e azimute (`az`) são convertidos de graus para radianos.
    - A distância em metros é convertida para radianos dividindo pelo raio da Terra.

5. Cálculo da nova latitude:

    - Utiliza fórmula esférica que considera a inclinação do azimute e a curvatura da Terra.

7. Cálculo da nova longitude:
    - Utiliza a fórmula que leva em conta o movimento no meridiano e a inclinação do azimute.

9. Conversão de volta para graus:
    - As novas coordenadas calculadas em radianos são convertidas para graus.

11. Retorno: Latitude e longitude do ponto final.

Retorno:

- Um par de valores:
    - `nova_lat`: Latitude do ponto final, em graus.
    - `nova_lon`: Longitude do ponto final, em graus.

Funções de Conversão de Unidades

khz_to_hz

```
# Converte MHz para Hz
def khz_to_hz(freq):
    return freq * 1e3
```

Converte kilohertz (kHz) para hertz (Hz).

- **Parâmetros**: `freq` (float/int): Frequência em kHz.
- **Retorno**: Frequência em Hz.

mhz_to_hz

```
# Converte MHz para Hz
def mhz_to_hz(freq):
    return freq * 1e6
```

Converte megahertz (MHz) para hertz (Hz).

- Parâmetros: `freq` (float/int): Frequência em MHz.
- Retorno: Frequência em Hz.

ghz_to_hz

```
# Converte MHz para Hz
def ghz_to_hz(freq):
    return freq * 1e9
```

Converte gigahertz (GHz) para hertz (Hz).

- Parâmetros: `freq` (float/int): Frequência em GHz.

- Retorno: Frequência em Hz.

dB_to_watts

```
# Converte dB para W
def dB_to_watts(dB):
    return 10 ** ((dB - 30) / 10)
```

Converte decibéis (dB) para watts (W).

- Parâmetros: `dB` (float): Valor em decibéis.
- Retorno: Potência em W.

dBm_to_watts

```
# Converte dBm para W
def dBm_to_watts(dBm):
    return 10**(dBm / 10) * 10**-3
```

Converte decibéis-milivatts (dBm) para watts (W).

- Parâmetros: `dBm` (float): Valor em dBm.
- Retorno: Potência em W.

dBm_to_mW

```
# Converte dBm para mW
def dBm_to_mW(dBm):
    return 10**(dBm/10)
```

Converte decibéis-milivatts (dBm) para milivatts (mW).

- Parâmetros: `dBm` (float): Valor em dBm.
- Retorno: Potência em mW.

watts_to_dBm

```
# Converte Watts para dBm
def watts_to_dBm(w):
    return 10 * log10(w) + 30
```

Converte watts (W) para decibéis-milivatts (dBm).

- Parâmetros: `w` (float): Potência em W.
- Retorno: Potência em dBm.

mW_to_dBm

```
# Converter mW para dBm
def mW_to_dBm(mW):
    return 10 * log10(mW)
```

Converte milivatts (mW) para decibéis-milivatts (dBm).

- Parâmetros: `mW` (float): Potência em mW.
- Retorno: Potência em dBm.

mW_to_watts

```
# Converter mW para W
def mW_to_watts(mW):
    return mW / 1000
```

Converte milivatts (mW) para watts (W).

- Parâmetros: `mW` (float): Potência em mW.
- Retorno: Potência em W.

watts_to_mW

```
# Converter W para mW
def watts_to_mW(w):
    return w * 1000
```

Converte watts (W) para milivatts (mW).

- Parâmetros: `watts` (float): Potência em W.
- Retorno: Potência em mW.

eirp_to_hz

```
# Transformando EIRP dBm em EIRP dBm/Hz
def eirp_to_hz(eirp):
    return eirp + 10 * log10(1 / 100000000)
```

Converte EIRP em dBm para EIRP por hertz (dBm/Hz).

- Parâmetros: `eirp` (float): EIRP em dBm.
- Retorno: Valor convertido em dBm/Hz.

largura_canal_hz

```
def largura_canal_hz(entrada):
    """
    Converte uma string com sufixo 'k', 'm' ou 'g' para um número float
    Args:
        entrada: String de entrada.
    Returns:
        float: Valor convertido, ou a entrada original se não for possí
    """

    # Converter para minúsculas e extrair os quatro primeiros caractere
    entrada_min = entrada[:4].lower()

    # Expressão regular para encontrar 'k', 'm' ou 'g' nos quatro prime
    padrao = r"[kmg]"
    match = re.search(padrao, entrada_min)
```

```
    if match:
        # Substituir a letra por ponto e converter para float
        indice_letra = match.start()
        novo_valor = float(entrada_min.replace(entrada_min[indice_letra
        if entrada_min[indice_letra] == 'k':
            return khz_to_hz(novo_valor)
        elif entrada_min[indice_letra] == 'm':
            return mhz_to_hz(novo_valor)
        else:
            return ghz_to_hz(novo_valor)
    else:
        # Se não encontrar 'k', 'm' ou 'g', retornar a entrada original
        return entrada
```

Converte uma string que representa frequência com sufixo (k, m, g) para hertz (Hz).

- Parâmetros: `entrada` (str): String contendo o valor e o sufixo.
- Retorno: Frequência em Hz como float, ou o valor original se não for convertível.