Trabalho de Matemática Computacional

Alunos:
Gabriel de Melo Osório – 107862
Henrique Shiguemoto Felizardo – 115207
Matheus Augusto Schiavon Parise – 107115

Professor:
Airton Marco Polidorio

Introdução

Este documento é uma coletânea de 3 trabalhos solicitados pelo professor Airton Marco Polidorio na disciplina de Matemática Computacional (6900-2021-T1) da Universidade Estadual de Maringá (UEM), os trabalhos requisitados são:

1. Fazer um algoritmo que calcule o seno de qualquer ângulo com precisão de 10 dígitos após a vírgula, utilizando a Série de Taylor e reduzindo a quantidade de multiplicações necessárias com o Esquema de Horner.
2. Calcular a raiz quadrada de um número positivo usando a infra-estrutura IEEE-754, utilizando o método de Newton Raphson.
3. Estimar a posição de um emissor de sinais por triangulação de sinais.

Trabalho 1

Abaixo estão as imagens referentes ao código implementado no trabalho 1

```python
import math
import numpy as np

def MCcos(x : float) -> float:
    # Mudança de variável para economizar multiplicações
    y = x*x

    # Constantes
    if2 = -0.5
    if4 = 1/24
    if6 = -1/720
    if8 = 1/40320
    if10 = -1/3628800
    if12 = 1/479001600

    # 7 primeiros termos da Série de Taylor para a função cosseno com reestruturação com esquema de Horner
    return (1 + y*(if2 + y*(if4 + y*(if6 + y*(if8 + y*(if10 + y*if12))))))

def MCsin(x : float) -> float:
    # Mudança de variável para economizar multiplicações
    y = x*x

    # Constante
    if3 = -1/6
    if5 = 1/120
    if7 = -1/5040
    if9 = 1/362880
    if11 = -1/39916800

    # 6 primeiros termos da Série de Taylor para a função seno com reestruturação com esquema de Horner
    return (x + x*y*(if3 + y*(if5 + y*(if7 + y*(if9 + y*if11)))))
```

```python
def main():
    PI = math.pi

    # Conversão do ângulo digitado pelo usuário em graus para radianos no intervalo entre [0, 2PI]
    # i = math.radians(float(input("Digite um ângulo (em graus): "))) % (2*math.pi)
    increment = 2*PI/30
    for i in np.arange(0.0, 2*PI, increment):
        # Conversão dos ângulos maiores que PI radianos para seu respectivo angulo negativo em radianos
        if (PI < i) and (i < 2*PI):
            i = -(2*PI - i)

        # Se estiver dentro do intervalo aceitável [-PI/4, PI/4], apenas calculamos o seno de i
        if (-PI/4 <= i) and (i <= PI/4):
            print("I = " + str(i) + " Erro = " + str(abs(MCsin(i) - math.sin(i))))

        # sen(x) = cos(x - PI/2) nesse intervalo
        elif (PI/4 < i) and (i <= 3*PI/4):
            print("I = " + str(i) + " Erro = " + str(abs(MCcos(i - PI/2) - math.sin(i))))

        # sen(x) = -sen(x - PI) nesse intervalo
        elif (3*PI/4 < i) and (i <= PI):
            print("I = " + str(i) + " Erro = " + str(abs(-MCsin(i - PI) - math.sin(i))))

        # sen(x) = -cos(x + PI/2) nesse intervalo
        elif (-3*PI/4 < i) and (i < -PI/4):
            print("I = " + str(i) + " Erro = " + str(abs(-MCcos(PI/2 + i) - math.sin(i))))

        # sen(x) = -sen(PI + x) nesse intervalo
        else:
            print("I = " + str(i) + " Erro = " + str(abs(-MCsin(PI + i) - math.sin(i))))


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

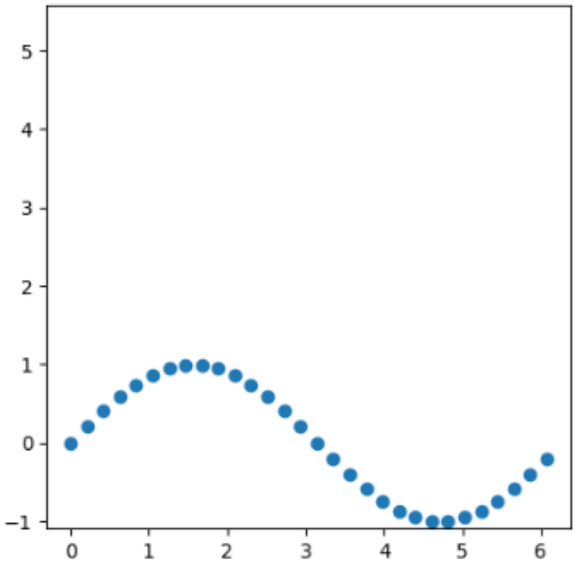
O código acima produz ângulos dentro de um for loop. Os testes são feitos com 30 ângulos diferentes, começando de 0 rad até 2*PI rad. Dependendo da posição que cada ângulo está no ciclo trigonométrico, uma regra trigonométrica diferente é utilizada. Quando o programa é executado, imprimimos o ângulo que está sendo testado e o erro (em módulo) do valor calculado pelo seno implementado pela nossa equipe e pelo seno da biblioteca math de Python. Escolhemos Python pela facilidade de desenvolver programas nele.

No intervalo [PI/4, -PI/4], apenas calculamos o seno do ângulo.
No intervalo (PI/4, 3*PI/4], calculamos o cos(ângulo - PI/2), que é igual a seno.
No intervalo (3*PI/4, PI], calculamos -sen(ângulo - PI), que é igual a seno.
No intervalo (-PI/4, -3*PI/4], calculamos -cos(ângulo - PI/2), que é igual a seno.
No intervalo (-3*PI/4, -PI), calculamos -sen(ângulo + PI), que é igual a seno.

Estamos realizando conversões para diminuir o valor do argumento das funções trigonométricas. Os cálculos de seno e cosseno são feitos utilizando uma aproximação por Séries de Taylor. No caso do seno, nós usamos 6 termos da série, enquanto que no cosseno nós utilizamos 7 termos. Além disso, transformamos a soma dos termos utilizando o Esquema de Horner para diminuir o número de multiplicações no geral.

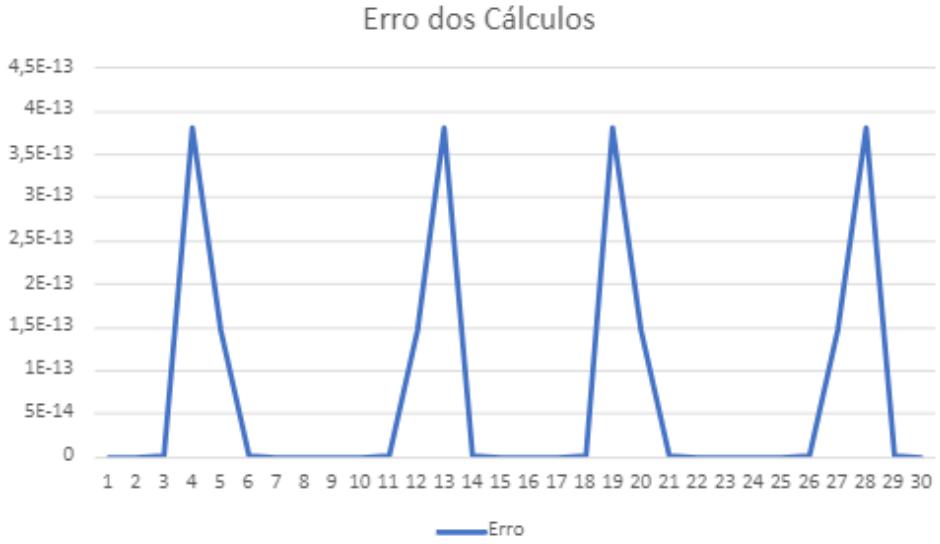
Gráfico

Para os testes deste trabalho foi solicitado dividir o ciclo de 360 graus em 30, e realizar testes com cada um dos intervalos. Esse gráfico representa os resultados observados dos testes do seno e cosseno do intervalo 0 radianos até 2π, com incremento de $2\pi/30$ por teste, totalizando 30 testes.



Resultados obtidos pelo algoritmo

Abaixo está o gráfico dos erros calculados no nosso programa, o eixo X é a iteração do loop na função mais e o eixo Y é o erro (em módulo):



Note que os erros crescem e decrescem quase harmonicamente. Além disso, os cálculos estão muito próximos dos valores reais, já que os erros ocorrem na 13° e 14° casa decimal. Demonstrando que as séries de Taylor, juntamente com o Esquema de Horner são um ótimo método para estimar valores de seno.

Trabalho 2

Abaixo está o código implementado do trabalho 2:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <math.h>

#define EXPONENT_SIZE 11
#define MANTISSA_SIZE 52
#define SQRT_2 1.414213562373095    //15 dígitos de precisão
#define PRECISION 0.0000000001      //verificar se o abs(erro) é menor que PRECISION

//União que pode ser vista como um número float em decimal (f) e também na forma IEEE754 (bits)
typedef union nIEEE{
    double f;
    struct{
        //ORDEM IMPORTA
        uint64_t mantissa : 52;
        uint64_t exponent : 11;
        uint64_t sign : 1;
    }bits;
} nIEEE;
```

```c
int main(void){
    //Testes
    for(float i = 0; i < 50; i = i + 0.7){

        //Inicialização da estrutura nIEEE
        nIEEE binary = {0};
        binary.f = i; //Definindo o valor de f (float decimal) e definindo os campos de bits (padrão IEEE-754)

        double valor_Newton_Raphson = newton_raphson(i, &binary);
        double valor_Sqrt_Math = sqrt(i);

        printf("i = %f - Erro = %.15lf\n", i, fabs(valor_Newton_Raphson - valor_Sqrt_Math));
    }
    return 0;
}
```

```c
double newton_raphson(double A, nIEEE* binary){

    //Caso a entrada do usuário for 0 ou 1, retornamos a própria entrada
    if(A == 0.0 || A == 1.0){
        return A;
    }
    //Em caso de números negativos como entrada, apenas temrinamos o programa
    else if (A < 0){
        printf("Erro! Argumento nao pode ser negativo!");
        exit(EXIT_FAILURE);
    }

    int64_t e = binary->bits.exponent - 1023; // Temos acesso direto ao expoente armazenado em padrão IEEE-754
    double f1 = 0; // (1 + f)

    //Definição dos valores (1 + f) e o valor que multiplicaremos a raiz de (1 + f)
    double resto_para_multiplicar = 1;
    if(e > 0){
        f1 = A / (2 << (e - 1));
        if((e % 2) == 0){
            resto_para_multiplicar = pow(2, e/2);
        }else{
            resto_para_multiplicar = SQRT_2*pow(2, (e-1)/2);
        }
    }else if(e < 0){
        f1 = A * (2 << (-e - 1));
        if((e % 2) == 0){
            resto_para_multiplicar = pow(2, e/2);
        }else{
            resto_para_multiplicar = (SQRT_2 / 2)*pow(2, (e+1)/2);
        }
    }else{
        f1 = A;
    }
```

```c
//Newton Raphson
double xk = ((f1 - 1) / 2) + 1;
double xk1 = (xk + (f1/xk))/2;
double erro = fabs(xk1 - xk);

while (erro > PRECISION)
{
    xk = xk1;
    xk1 = (xk + (f1/xk))/2;
    erro = fabs(xk1 - xk);
}

return xk1*resto_para_multiplicar;
```

Com o uso de estruturas de dados chamadas de uniões, presentes nas linguagens C/C++, nós podemos criar uma struct de campos de bits (52 bits para mantissa, 11 para o expoente e 1 para o sinal) e também criar uma variável para representar nosso número de ponto flutuante de 64 bits (o tipo dessa variável é double, pois float em C possui 32 bits). Se essas duas estruturas pertencerem a uma união, então as mesmas compartilharão o mesmo espaço de memória. Isso significa que, se definirmos o nosso valor double (no código se chama f) com um valor qualquer, os campos da struct nIEEE automaticamente terão os valores de f convertidos para o padrão IEEE-754. Esse foi o principal motivo para a escolha da nossa linguagem para implementar este segundo trabalho ser C, pois usando

uniões juntamente com campos de bits, temos acesso a todos os 64 bits armazenados para um número do tipo double.

Na função main, temos os testes. Os testes consistem num for loop em que o contador é um do tipo float e incrementamos o contador pôr 0.7 em cada iteração (o valor de incremento podia ser qualquer um, mas escolhemos um valor que nos garantisse valores que não fossem apenas números inteiros). Chamamos as funções de raiz quadrada implementadas pela equipe (newton_raphson) e sqrt() da biblioteca math.h e calculamos o erro entre os dois retornos e imprimimos 15 casas decimais desse erro (o erro está em módulo).

A função de newton_raphson começa de maneira simples, se a entrada for 0 ou 1, então devolvemos a própria entrada ($\sqrt{0} = 0$ e $\sqrt{1} = 1$). Se a entrada for negativa, o programa imprime uma mensagem de erro e apenas encerra a execução com código EXIT_FAILURE. Nos casos restantes, calculamos o valor do expoente da potência de 2 (variável e). Calculamos o valor de 1 + f (variável f1, valor que representa o valor do argumento reduzido da raiz quadrada) juntamente com o valor que multiplicaremos $\sqrt{(1+f)}$ para conseguir a raiz quadrada da entrada de volta. Com esses valores podemos começar o algoritmo de newton_raphson calculando o chute inicial (variável xk) e já podemos calcular o valor de x_{k+1} (variável xk1), daí num loop while podemos ir recalculando os valores de x_k e x_{k+1} até que o valor de xk+1 alcance a precisão (#define PRECISION, no início do código). Com isso conseguimos retornar a nossa aproximação da raiz quadrada.

Abaixo segue o gráfico dos erros obtidos para os testes realizados na função main. O eixo X representa o ângulo enquanto que o eixo Y é o erro entre o valor calculado pela função MCsin e a função sin da math.h.

É possível notar que os valores calculados pela função newton_raphson são muito precisos, na maioria dos ângulos os erros foram de 0, ou seja, a função acertou 15 dígitos decimais. E em alguns casos, a função acertou 14 dígitos. Isso demonstra que o newton_raphson é um ótimo método para cálculo de raíz quadrada, com uma precisão ótima em poucas iterações se considerarmos a redução do valor de argumento, utilizando a representação de ponto flutuante no sistema IEEE-754.

Trabalho 3

1) Os erros que podem comprometer esses dados são:

Erro no modelo: exemplo de erro no modelo é a série de taylor pois não é possível calcular até o infinito, então devemos escolher a margem de erro calculando até um termo x, que definirá até que casa decimal o resultado será confiável.

Erro nos dados: Há várias formas e interferência na captura dos dados do sinal como por exemplo: a camada de ozônio, a camada eletromagnética, poeira, etc.

Erro de truncamento e arredondamento: este erro já é tratado pelo computador automaticamente, é a capacidade de decidir quando um valor como:

⅓ = 0.333 … será tratado, existem duas formas:

Arredondar ou Truncar, a primeira aumenta o valor da última casa decimal para o maior número superior a este, a segunda apenas descarta o resto da equação, então considerando um computador que aguenta 8 casas decimais depois da vírgula:

Arredondamento:
⅓ = 0.33333333… = 0.33333334

Truncamento:
⅓ = 0.33333333… = 0.33333333

2) Abaixo está o código implementado em Python para a resolução do problema de Triangulação de Sinais:

```python
import numpy as np

class Receptor:
    k: int
    x: float
    y: float
    p0k: float
    Lk: float

    def Criar(self, k, x, y, p0k, Lk):
        self.k = k
        self.x = x
        self.y = y
        self.p0k = p0k
        self.Lk = Lk
        return self

    def Print(self):
        print("k:", self.k,"x:", self.x,"y:", self.y,"p0k:", self.p0k,"Lk:", self.Lk)

# - estimativa de distância radial que uma fonte emissora está do receptor k (em metros) em função da
# força do sinal pk que ele recebe.
def DistanciaK(Recp: Receptor, Pk: float):
    distancia = 10**((Recp.p0k-Pk)/(10*Recp.Lk))
    return distancia
```

```python
def Exercicio1():
    print("\nExercicio 1:")

    #Constantes
    PosicaoReal = np.array([[0.0], [9.0]])
    pk1 = -48.4
    pk2 = -50.6
    pk3 = -32.2
    pk4 = -47.4
    pk5 = -46.3

    print("\nConsiderando os Receptores:")
    Recp1 = Receptor().Criar(1, 1.55, 17.63, -26.0, 2.1)
    Recp2 = Receptor().Criar(2, -4.02, 0.00, -33.8, 1.8)
    Recp3 = Receptor().Criar(3, -4.40, 9.60, -29.8, 1.3)
    Recp4 = Receptor().Criar(4, 9.27, 4.64, -31.2, 1.4)
    Recp5 = Receptor().Criar(5, 9.15, 12.00, -33.0, 1.5)

    #Cálculos dos raios das circunferências
    d1 = DistanciaK(Recp1, pk1)
    d2 = DistanciaK(Recp2, pk2)
    d3 = DistanciaK(Recp3, pk3)
    d4 = DistanciaK(Recp4, pk4)
    d5 = DistanciaK(Recp5, pk5)

    #Matriz de Coeficientes
    A = np.array([[2 * (Recp5.x - Recp1.x), 2 * (Recp5.y - Recp1.y)],
                  [2 * (Recp5.x - Recp2.x), 2 * (Recp5.y - Recp2.y)],
                  [2 * (Recp5.x - Recp3.x), 2 * (Recp5.y - Recp3.y)],
                  [2 * (Recp5.x - Recp4.x), 2 * (Recp5.y - Recp4.y)]])
```

```python
#Matriz de Resultados
B = [[(d1*d1) - (d5*d5) - (Recp1.x*Recp1.x) - (Recp1.y*Recp1.y) + (Recp5.x*Recp5.x) + (Recp5.y*Recp5.y)],
     [(d2*d2) - (d5*d5) - (Recp2.x*Recp2.x) - (Recp2.y*Recp2.y) + (Recp5.x*Recp5.x) + (Recp5.y*Recp5.y)],
     [(d3*d3) - (d5*d5) - (Recp3.x*Recp3.x) - (Recp3.y*Recp3.y) + (Recp5.x*Recp5.x) + (Recp5.y*Recp5.y)],
     [(d4*d4) - (d5*d5) - (Recp4.x*Recp4.x) - (Recp4.y*Recp4.y) + (Recp5.x*Recp5.x) + (Recp5.y*Recp5.y)]]

#Matriz Transposta de A
A_t = A.transpose()

X = np.matmul(np.matmul(np.linalg.inv(np.matmul(A_t, A)), A_t), B)
Erro = np.absolute(np.subtract(X, PosicaoReal))

print(X)
print(Erro)
```

```python
def Exercicio2():
    print("\nExercicio 2:")
    PosicaoReal = np.array([[3.0], [3.0]])
    pk1 = -46.9
    pk2 = -46.4
    pk3 = -41.2
    pk4 = -45.8
    pk5 = -48.7

    print("\nConsiderando os Receptores:")
    Recp1 = Receptor().Criar(1, 1.55, 17.63, -26.0, 2.1)
    Recp2 = Receptor().Criar(2, -4.02, 0.00, -33.8, 1.8)
    Recp3 = Receptor().Criar(3, -4.40, 9.60, -29.8, 1.3)
    Recp4 = Receptor().Criar(4, 9.27, 4.64, -31.2, 1.4)
    Recp5 = Receptor().Criar(5, 9.15, 12.00, -33.0, 1.5)

    #Cálculos dos raios das circunferências
    d1 = DistanciaK(Recp1, pk1)
    d2 = DistanciaK(Recp2, pk2)
    d3 = DistanciaK(Recp3, pk3)
    d4 = DistanciaK(Recp4, pk4)
    d5 = DistanciaK(Recp5, pk5)

    #Matriz de Coeficientes
    A = np.array([[2 * (Recp5.x - Recp1.x), 2 * (Recp5.y - Recp1.y)],
                  [2 * (Recp5.x - Recp2.x), 2 * (Recp5.y - Recp2.y)],
                  [2 * (Recp5.x - Recp3.x), 2 * (Recp5.y - Recp3.y)],
                  [2 * (Recp5.x - Recp4.x), 2 * (Recp5.y - Recp4.y)]])

    #Matriz de Resultados
    B = [[(d1*d1) - (d5*d5) - (Recp1.x*Recp1.x) - (Recp1.y*Recp1.y) + (Recp5.x*Recp5.x) + (Recp5.y*Recp5.y)],
         [(d2*d2) - (d5*d5) - (Recp2.x*Recp2.x) - (Recp2.y*Recp2.y) + (Recp5.x*Recp5.x) + (Recp5.y*Recp5.y)],
         [(d3*d3) - (d5*d5) - (Recp3.x*Recp3.x) - (Recp3.y*Recp3.y) + (Recp5.x*Recp5.x) + (Recp5.y*Recp5.y)],
         [(d4*d4) - (d5*d5) - (Recp4.x*Recp4.x) - (Recp4.y*Recp4.y) + (Recp5.x*Recp5.x) + (Recp5.y*Recp5.y)]]
```

```python
    #Matriz Transposta de A
    A_t = A.transpose()

    X = np.matmul(np.matmul(np.linalg.inv(np.matmul(A_t, A)), A_t), B)
    Erro = np.absolute(np.subtract(X, PosicaoReal))

    print(X)
    print(Erro)


def main():
    Exercicio1()
    Exercicio2()

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

Primeiramente, criamos uma classe para representar os Receptores (de 1 à 5) que foram dados no problema. Essa classe consiste de um Construtor que apenas inicializa as variáveis como posição x, posição y, seu id (variável k), a potência de referência do k-ésimo receptor medido a 1 metro de distância da fonte de sinal (p0k) e o fator de atenuação para o k-ésimo receptor (Lk). Também temos um método de impressão de dados do receptor (Print) apenas para debug. Com essa classe conseguimos criar 5 receptores para resolver os casos 1 e 2 dados no trabalho. Separamos os dois casos como duas funções diferentes. Ou seja, quando chamamos a função Exercicio1(), o primeiro caso do trabalho em que o emissor está na posição (0.0, 9.0) é resolvido.

Para ambos os casos, definimos os receptores de 1 à 5 e também definimos os pk1, pk2, pk3, pk4 e pk5 (valores dados no enunciado do problema) para calcular os valores de estimativa de distância radial dk, para este cálculo implementamos uma função que realiza o cálculo dado no enunciado do trabalho, essa função se chama DistanciaK(). Segue a fórmula de cálculo de estimativa de distância radial:

$$d_k = 10^{\frac{\rho_0^k - \rho^k}{10\mathcal{L}^k}}$$

Precisamos dessas distâncias para montar o sistema de equações para estimar a posição do emissor, as estimativas foram chamadas de d1, d2, d3, d4 e d5, correspondentes à estimativa radial de cada receptor, respectivamente.
Para calcular de fato uma estimativa de posição do emissor, precisamos montar um sistema de equações. Para isso, criamos a matriz A (que é apenas uma matriz de coeficientes que multiplicam os valores de x e y do emissor), a matriz B (que é a apenas uma matriz de resultados, ou seja, A.X = B, em que X é uma matriz com a posição do emissor) e a Transposta de A (A_t, que é usada para isolar a matriz resultante X).

Nosso sistema de equações possui 5 equações e 2 variáveis inicialmente, as equações correspondem às equações de circunferência para cada um dos receptores em que os raios são as variáveis d1, d2, d3, d4 e d5, e as 2 variáveis correspondem à posição x e y do emissor, que queremos estimar. Porém, nós realizamos uma mudança no sistema em questão para conseguir um sistema equivalente, isso se deve ao fato de que as equações que temos não são lineares, mas sim quadráticas, que são mais difíceis de resolver. Desse modo, escolhemos a última equação para subtrairmos de todas as outras equações. Daí temos um sistema de equações lineares com 4 equações e 2 variáveis.

Como precisamos isolar a matriz com a posição do emissor, que chamamos de X no código, temos que tentar realizar esse isolamento a partir da seguinte equação, A.X

= B, em que A é a matriz de coeficientes, X é a matriz de estimativa da posição do emissor e B é a matriz de resultados, temos o seguinte desenvolvimento:

$$A.X = B$$

$$\Rightarrow \quad A\_t.A.X = A\_t.B$$

$$\Rightarrow \quad inversa(A\_t.A).A\_t.A.X = inversa(A\_t.A).A\_t.B$$

$$\Rightarrow \quad X = inversa(A\_t.A).A\_t.B$$

Chegamos ao resultado de que para calcular a matriz X, precisamos da matriz B, da matriz A, da matriz transposta de A (A_t) e da matriz inversa da multiplicação entre a transposta de A (A_t) e A. A ordem da multiplicação importa, pois a multiplicação de matrizes não é comutativa.
Para realizar as operações entre matrizes, usamos o módulo numpy do Python, conseguimos realizar as operações de multiplicação, transposta e a inversa de uma matriz quadrada.

Realizando as multiplicações necessárias, temos o resultado que estamos esperando, que é a estimativa de posição do emissor. No nosso programa, nós imprimimos as estimativas de posição do emissor e também o valor absoluto dos erros da estimativa se comparados com a posição real dos emissores em cada caso.

Receptores

Receptor k	Coordenada (metros) do Receptor k			ρ_0^k – Potência de referência do receptor k (em dBm a 1 metro do receptor)	$\mathcal{L}^k$ – Fator de atenuação do receptor k
	x	y	z		
1	1.55	17.63	1.35	-26.0	2.1
2	-4.02	0.00	1.35	-33.8	1.8
3	-4.40	9.60	1.35	-29.8	1.3
4	9.27	4.64	1.35	-31.2	1.4
5	9.15	12.00	1.35	-33.0	1.5

Emissor 1

Receptor k	1	2	3	4	5
ρ^k(dBm)	-48.4	-50.6	-32.2	-47.4	-46.3

Emissor 2

Receptor k	1	2	3	4	5
ρ^k(dBm)	-46.9	-46.4	-41.2	-45.8	-48.7

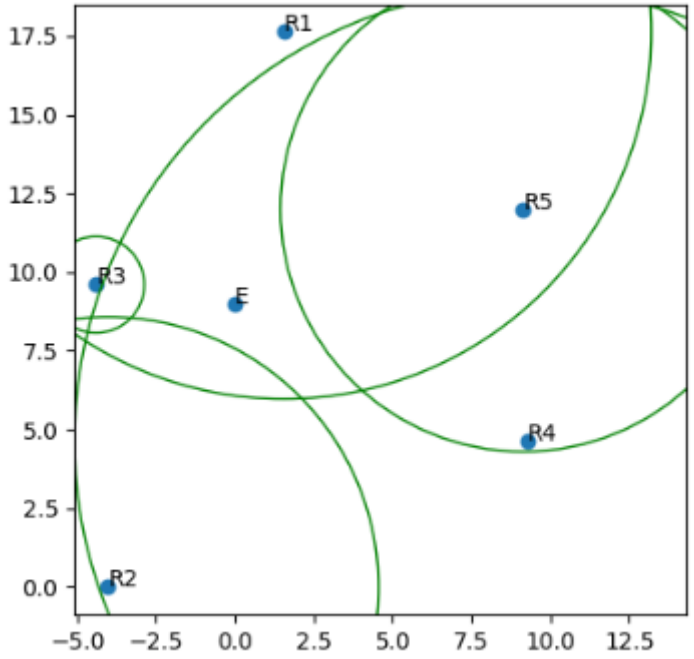
Resultados

Os resultados obtidos para o exercício 1 são X = 0.99252613 e Y = 9.71044952, com erro de 0.99252613 para o X e 0.71044952 para o Y. Para o exercício 2, X = -0.87386104 e Y = 6.84236916, com erro de 3.87386104 para X e 3.84236916 para Y.

Gráficos

O documento de especificação contém os dados dos receptores e dos emissores e apresenta dois casos a serem solucionados pela implementação do trabalho.

Posições dos Receptores e Emissor no Caso 1



Posições dos Receptores e Emissor no Caso 2