Nome	**NUSP**
Felipe Caracciolo Gonçalves	8988341
Rodrigo Dias Manduca	9017269

Exercício-Programa conjunto MAP3121 / PEA3301 -
Fluxo de Potência em redes elétricas pelo Método de Newton

Para resolver esse problema foi necessário primeiro definir os vetores e matrizes, bem como seus valores. Para tanto começa-se com os valores obtidos de arquivos.

A fim de tornar mais receptivo na decisão de qual caso analisar (mínimo de alterações no código), cria-se uma função que faz a definição dos arquivos, conforme mostra a Figura 1.

```
void decideRede(char *barra, char *nodal) {
    char *p, s[100];
    int n;
    printf("Decida a Rede\n");
    printf("    |-> 1) Stevenson\n");
    printf("    |-> 2) Reticulada\n");
    printf("    |-> 3) Distribiucao Primaria\n");
    printf("    |-> 4) Distribiucao Primaria Secundaria\n");
    printf("Favor escolha uma opcao: ");
    //pega do teclado
    while (fgets(s, sizeof(s), stdin)) {
        n = strtol(s, &p, 10);
        // caso não seja int
        if ((p == s || *p != '\n')) {
            printf("Favor escolha uma opcao: ");
        }
        // não ser opção válida
        else if (n != 1 && n!= 2 && n!= 3 && n!=4) {
            printf("Favor escolha uma opcao: ");
        }
        // sai do loop
        else break;
    }
    if (n == 1) {
        strcpy(barra, "Redes/1_Stevensonn/1_Stevensonn_DadosBarras.txt");
        strcpy(nodal, "Redes/1_Stevensonn/1_Stevensonn_Ynodal.txt");
    }
    else if (n == 2) {
        strcpy(barra, "Redes/2_Reticulada/2_Reticulada_DadosBarras.txt");
        strcpy(nodal, "Redes/2_Reticulada/2_Reticulada_Ynodal.txt");
    }
    else if (n == 3) {
        strcpy(barra, "Redes/3_DiPrimaria/3_DiPrimaria_DadosBarras.txt");
        strcpy(nodal, "Redes/3_DiPrimaria/3_DiPrimaria_Ynodal.txt");
    }
    else {
        strcpy(barra, "Redes/4_Secundaria/4_Secundaria_DadosBarras.txt");
        strcpy(nodal, "Redes/4_Secundaria/4_Secundaria_Ynodal.txt");
    }
}
```

Figura 1 - Função que define os arquivos do caso a analisar

Em seguida lê-se cada arquivo. O arquivo que contém os dados das barras é lido duas vezes. Na primeira (Figura 2) obtém-se os valores fixos e necessários, sendo eles: o número total de barras; n_1 (quantia de barras tipo PQ); e n_2 (quantia de barras tipo PV).

```
void leNumeroDeLinhas(FILE *file, char *barra, int *nBarras, int *nPQ, int *nPV)
{
    int i, linha;
    double tipo, par1, par2, tensao;

    fopen_s(&file, barra, "r");
    if (file == NULL)
    { // Open source file.
        perror("fopen source-file");
        return;
    }

    fscanf_s(file, "%d", nBarras);
    *nPQ = 0;
    *nPV = 0;

    for (i = 0; i < *nBarras; i++) {
        fscanf_s(file, "%d", &linha);
        fscanf_s(file, "%lf", &tipo);
        fscanf_s(file, "%lf", &tensao);
        fscanf_s(file, "%lf", &par1);
        fscanf_s(file, "%lf", &par2);
        if(tipo == 0){
            *nPQ = *nPQ + 1;
        }
        else if(tipo == 1){
            *nPV = *nPV + 1;
        }
    }

    fclose(file);
}
```

Figura 2 - Primeira leitura do arquivo de barras

O número de barras define o tamanho dos vetores de tensão (módulo e ângulo). n_2 define o tamanho do vetor de potência ativa (de geração), n_1+n_2 o tamanho do vetor auxiliar J, o qual marca quais os números das barras PQ e PV (nesta ordem).

A segunda leitura desse tipo de arquivo (Figura 3 a) e a de arquivo com as admitâncias dos trechos (Figura 3 b) pega os dados dos arquivos e coloca na matriz correspondente.

```c
void leDadosBarra(FILE *file, double **matriz,  char *barraPath)
{
    int numeroBarras = 0;
    int i, j;
    int barraNumero;

    fopen_s(&file, barraPath, "r");
    if (file == NULL)
    { // Open source file.
        perror("fopen source-file");
        return;
    }

    fscanf_s(file, "%d", &numeroBarras);

    for (i = 0; i < numeroBarras; i++) {
        fscanf_s(file, "%d", &barraNumero);
        for (j = 0; j < colunaBarra; j++) {
            fscanf_s(file, "%lf", &matriz[i][j]);
        }
    }

    fclose(file);
}
```

Figura 3 a - Leitura de dados das Barras

```c
void leDadosTrecho(FILE *file, double **matrizG, double **matrizB, char *nodalPath)
{
    int numeroTrechos = 0;
    int i, j, linha, coluna;

    fopen_s(&file, nodalPath, "r");
    if (file == NULL)
    { // Open source file.
        perror("fopen source-file");
        return;
    }

    fscanf_s(file, "%d", &numeroTrechos);

    for (i = 0; i < numeroTrechos; i++) {
        fscanf_s(file, "%d", &linha);
        fscanf_s(file, "%d", &coluna);
        fscanf_s(file, "%lf", &matrizG[linha][coluna]);
        fscanf_s(file, "%lf", &matrizB[linha][coluna]);
    }

    fclose(file);
}
```

Figura 3 b - Leitura de admitâncias dos trechos

Figura 3 - Funções de leitura de dados de arquivos

Com os valores lidos, pode-se popular os vetores anteriormente descritos com seus valores inicias. Da especificação do problema tem-se que as matrizes e vetores coluna utilizados no método de Newton tem tamanho ($2*n_1 + n_2$). Da mesma também tem-se como calcular o valor da matriz Jacobiana que, na implementação foi agrupada em dois grupos: o primeiro (Figura 4 a) calcula os valores de $\frac{df_p}{d\theta}$ e $\frac{df_p}{dV}$ e o segundo (Figura 4 b) calcula $\frac{df_q}{d\theta}$ e $\frac{df_q}{dV}$, ambos conforme especificado no enunciado do problema.

```
// Cálculo de dfp (dfp/dtheta e dfp/dV)
for (int i = 0; i < nPQ+nPV; i++)
{
    int j = jIndice[i];
    double dtheta;
    for (int c = 0; c < 2*nPQ + nPV; c++)
    {
        if(c == i) { // Eq (13) 1 -> dfpj/dthetaj
            dtheta = 0;
            for (int k = 0; k < tamanho; k++)
            {
                double thetaKJ;
                if(k != j) {
                    thetaKJ = fase[k] - fase[j];
                    dtheta += tensao[k]*(matrizG[j][k]*sin(thetaKJ) + matrizB[j][k]*cos(thetaKJ));

                }
            }
            del[i][c] = tensao[j]*dtheta;
        }
        else if (c == nPV + nPQ + i) // Eq (13) 2 -> dfpj/dVj
        {
            dtheta = 0;
            for (int k = 0; k < tamanho; k++)
            {
                double thetaKJ;
                if(k != j) {
                    //printf("k = %d\n", k);
                    thetaKJ = fase[k] - fase[j];
                    dtheta+= tensao[k]*(matrizG[j][k]*cos(thetaKJ) - matrizB[j][k]*sin(thetaKJ));
                }
            }
            del[i][c] = dtheta + 2*tensao[j]*matrizG[j][j];
        }
        else if (c < nPV + nPQ) // Eq (13) 3 -> dpfj/dthetak
        {
            int k = (int)jIndice[c];
            double thetaKJ = fase[k] - fase[j];
            del[i][c] = (-1)*tensao[j]*tensao[k]*(matrizG[j][k]*sin(thetaKJ) + matrizB[j][k]*cos(thetaKJ));
        }
        else // Eq (13) 4 -> dfpj/dVk
        {
            int k = (int)jIndice[c - nPQ - nPV];
            double thetaKJ = fase[k] - fase[j];
            del[i][c] = tensao[j]*(matrizG[j][k]*cos(thetaKJ) - matrizB[j][k]*sin(thetaKJ));
        }
    }
}
```

Figura 4 a - Cálculo de dfp/dϑ e dfp/dV

```
// Cálculo de dfq (dfq/dtheta e dfq/dV)
for (int i = 0; i < nPQ; i++)
{
    int j = jIndice[i];
    double dtheta;
    for (int c = 0; c < 2*nPQ + nPV; c++)
    {
        if(c == i) { // Eq (14) 1 -> dfqj/dthetaj
            dtheta = 0;
            for (int k = 0; k < tamanho; k++)
            {
                if(k != j) {
                    double thetaKJ = fase[k] - fase[j];
                    dtheta+= tensao[k]*(matrizG[j][k]*cos(thetaKJ) - matrizB[j][k]*sin(thetaKJ));
                }
            }
            del[nPQ+nPV+i][c] = tensao[j]*dtheta;
        }

        else if (c == nPV + nPQ + i) // Eq (14) 2 -> dfqj/dVj
        {
            dtheta = 0;
            for (int k = 0; k < tamanho; k++)
            {
                if(k != j) {
                    double thetaKJ = fase[k] - fase[j];
                    dtheta+= tensao[k]*(matrizG[j][k]*sin(thetaKJ) + matrizB[j][k]*cos(thetaKJ));
                }
            }
            del[nPQ+nPV+i][c] = -dtheta - 2*tensao[j]*matrizB[j][j];
        }

        else if (c < nPV + nPQ) // Eq (14) 3 -> dfqj/dthetak
        {
            int k = (int)jIndice[c];
            double thetaKJ = fase[k] - fase[j];
            del[nPQ+nPV+i][c] = (-1)*tensao[j]*tensao[k]*(matrizG[j][k]*cos(thetaKJ) - matrizB[j][k]*sin(thetaKJ));
        }

        else // Eq (14) 4 -> dfpj/dVk
        {
            int k = (int)jIndice[c - nPQ - nPV];
            double thetaKJ = fase[k] - fase[j];
            del[nPQ+nPV+i][c] = (-1)*tensao[j]*(matrizG[j][k]*sin(thetaKJ) + matrizB[j][k]*cos(thetaKJ));
        }
    }
}
```

Figura 4 b - Cálculo d dfq/dϑ e dfq/dV

Figura 4 - Cálculo do jacobiano

O passo seguinte foi determinar os valores do vetor coluna de desvio de potência (cálculo também definido no enunciado do problema). Dessa forma, implementou-o conforme a **Erro! Fonte de referência não encontrada.**.

```
// Cálculo de fpj
for (int i = 0; i < nPQ+nPV; i++)
{
    int j = jIndice[i];
    double fp = 0;
    for (int k = 0; k < tamanho; k++)
    {
        double thetaKJ = fase[k] - fase[j];
        fp += tensao[k]*(matrizG[j][k]*cos(thetaKJ) - matrizB[j][k]*sin(thetaKJ));
    }
    f[i][0] = tensao[j]*fp;
}

// Como os nPV ultimos correspondem às barras PV, em que a potência especificada não é necessariamente nula
for (int i = 0; i < nPV; ++i)   {   f[nPQ+i][0] = f[nPQ+i][0] - pEsp[i];    }

// Cálculo de fqj
for (int i = 0; i < nPQ; i++)
{
    int j = jIndice[i];
    double fq = 0;
    for (int k = 0; k < tamanho; k++)
    {
        double thetaKJ = fase[k] - fase[j];
        fq += tensao[k]*(matrizG[j][k]*sin(thetaKJ) + matrizB[j][k]*cos(thetaKJ));
    }
    f[nPQ+nPV+i][0] = (-1)*tensao[j]*fq;
}
```

Figura 5 - Cálculo do valor do vetor de desvio de potência

O passo seguinte é a resolução através do método de Newton. A primeira etapa foi a resolução, para tal é necessário calcular o inverso da matriz. Para isso se calcula o determinante da matriz, conforme apresenta a Figura 6.

```
double determinant(double **a, int tamanho)
{
    int i,j,j1,j2;
    double det = 0;
    double **m = NULL;
    if (tamanho < 1) { /* Error */

    } else if (tamanho == 1) { /* Shouldn't get used */
        det = a[0][0];
    } else if (tamanho == 2) {
        det = a[0][0] * a[1][1] - a[1][0] * a[0][1];
    } else {
        det = 0;
        for (j1=0;j1<tamanho;j1++) {
            m = (double **)malloc((tamanho-1)*sizeof(double *));
            for (i=0;i<tamanho-1;i++)
                m[i] = (double *)malloc((tamanho-1)*sizeof(double));
            for (i=1;i<tamanho;i++) {
                j2 = 0;
                for (j=0;j<tamanho;j++) {
                    if (j == j1)
                        continue;
                    m[i-1][j2] = a[i][j];
                    j2++;
                }
            }
            det += pow(-1.0,1.0+j1+1.0) * a[0][j1] * determinant(m,tamanho-1);
            for (i=0;i<tamanho-1;i++)
                free(m[i]);
            free(m);
        }
    }
    return(det);
}
```

Figura 6 - Cálculo de determinante

Com esse método pode-se calcular a matriz de cofatores do matriz Jacobiana (Figura 7) e então invertê-la (Figura 8).

```
void cofactor(double **matriz, double **inversa, int tamanho, int tamanhoOriginal)
{
    double **b = inicializa_Matrix(tamanhoOriginal, tamanhoOriginal);
    double **fac = inicializa_Matrix(tamanhoOriginal, tamanhoOriginal);
    int p,q,m,n,i,j;
    for (q=0;q<tamanho;q++)
    {
        for (p=0;p<tamanho;p++)
        {
            m=0;
            n=0;
            for (i=0;i<tamanho;i++)
            {
                for (j=0;j<tamanho;j++)
                {
                    if (i != q && j != p)
                    {
                        b[m][n]=matriz[i][j];
                        if (n<(tamanho-2))
                            n++;
                        else
                        {
                            n=0;
                            m++;
                        }
                    }
                }
            }
            fac[q][p]=(double)pow(-1,q + p) * determinant(b,tamanho-1);
        }
    }
    inverter(matriz,fac, inversa, tamanho, tamanhoOriginal);
}
```

Figura 7 - Cálculo da matriz de cofatores

```c
/*Finding inverter of matrix*/
void inverter(double **matriz,double **fac, double **inversa, int tamanho, int tamanhoOriginal)
{
    int i,j;
    double **b = inicializa_Matrix(tamanhoOriginal, tamanhoOriginal);
    double d;

    for (i=0;i<tamanho;i++)
    {
        for (j=0;j<tamanho;j++)
        {
            b[i][j]=fac[j][i];
        }
    }
    d=determinant(matriz,tamanho);
    for (i=0;i<tamanho;i++)
    {
        for (j=0;j<tamanho;j++)
        {
            inversa[i][j]=b[i][j] / d;
        }
    }
}
```

Figura 8 - Termina o processo de inverter a matriz jacobiana (transpõe a dos cofatores e divide pelo determinante)

Faz-se isso pelo método de Newton utilizar matrizes, sendo a Jacobiana uma quadrada. Então é possível invertê-la e tem-se que:

$$Jacob \cdot [\Delta] = [desvio]$$

$$[Jacob]^{-1} \cdot Jacob \cdot [\Delta] = [Jacob]^{-1} \cdot [desvio]$$

$$[\Delta] = [Jacob]^{-1} \cdot [desvio]$$

Com isso e sabendo o valor da inversa da matriz jacobiana tudo que resta é calcular a multiplicação dessas matrizes (implementação na Figura 9).

```c
double ** multiplyMatrix(int m1, int m2, double **matrizA,
                         int n1, int n2, double **matrizB)
{
    int x, i, j;
    double **res = inicializa_Matrix(m1,  n2);
    for (i = 0; i < m1; i++) {
        for (j = 0; j < n2; j++) {
            res[i][j] = 0;
            for (x = 0; x < m2; x++) {
                *(*(res + i) + j) += *(*(matrizA + i) + x) *
                                     *(*(matrizB + x) + j);
            }
        }
    }
    return res;
}
```

Figura 9 - Cálculo de multiplicação de matrizes

Tendo-se o valor de uma iteração, o que se fez foi tornar o método de resolução do método de Newton iterativo, atualizando o valor dos vetores de tensão e ângulos com os obtidos como resultado ([Δ]) até que este seja menor que um determinado valor relativo (assumiu-se 10%) do valor atual (de cada tensão e ângulo).

Com os valores de tensão e ângulo de cada barra e as admitâncias (do arquivo) foi possível se obter os valores desejados.

Apresentação dos Resultados

Observação: dos casos 3 e 4 não serão apresentados pois pela implementação (matriz como vetor de vetores houve estouro nas posições de memória na RAM devido ao tamanho das matrizes, sendo que são necessárias, mesmo que se calcule apenas alguns pontos. Uma possível solução que não chegou a ser implementada seria utilizar um vetor único e trabalhar em cima disso, dado que se sabe o tamanho de uma linha – nl - e o elemento das colunas seguintes seriam – nl*i +j – em que i representa a linha e j a coluna).

Rede 1

Barra	Tensão Complexa		Módulo da tensão complexa (V)
	Módulo (pu)	Ângulo (°)	
0	1.000000	0	132790.562
1	0.939973	-5.291	124819.528
2	1.000000	-2.396	132790.562
3	0.912165	-8.978	121126.928
4	0.945753	-5.247	125587.063

Trecho		Potência ativa (kW)	Potência reativa (kVAr)	Perda ativa (kW)
Barra inicial	Barra final			
0	1	57549.621	21691.446	1628.115
0	4	75463.782	26143.584	2004.567
1	2	-45686.412	-33973.860	1106.246
2	3	35990.462	14913.627	1387.623
2	4	27313.614	16981.034	598.652
3	4	-23710.325	-8075.639	450.320

	Valor (kW)
Potência total gerada	378210.234
Potência total absorvida	371035.153
Potência total perdida	7175.523