2º Prova de Implementação de ALC
2016.2

Professor: Erito

Alunos: Edson Neves
Maria Eduarda
Felipe Rangel
Willian Amphilophio
Vitor Loura
Victor Baessa
Gabriel Marques

Questão 3 – PSO

Main.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#include "matrizlib.h"

// Gera uma matriz aleatória com elementos variando de -max a +max
Matriz Matriz_initAleatoria(unsigned int linhas, unsigned int colunas, int max)
{
	Matriz matriz;

	matriz.linhas = linhas;
	matriz.colunas = colunas;
	matriz.triangular = NAO;

	int i, j;

	if ( !( matriz.valor = (double**)malloc(linhas*sizeof(double*)) ) ) {
		if (Matriz_debug == 1) printf ("Nao foi possivel criar matriz\n");
	}
	for ( i = 0; i < linhas; i++) {
		if ( !( matriz.valor[i] = (double*)malloc(colunas*sizeof(double) ) ) ) {
			if (Matriz_debug == 1) printf ("Nao foi possivel criar matriz\n");
		}
		else
			for( j = 0; j < colunas; j++ )
				matriz.valor[i][j] = rand() % ( ( max + 1 ) * 2 ) - max;
	}

	return matriz;
}

Matriz Matriz_residuo( Matriz A, Matriz x, Matriz b )
{
	Matriz Ax = Matriz_mult( A, x, 0 );
	Matriz b_Ax = Matriz_sub( b, Ax, 0 );
	Matriz_free( Ax );

	return b_Ax;
}

Matriz PSO( Matriz A, Matriz b, double max_aleatorio )
{
	int qnt = ceil( A.colunas * 0.3 );
	int i, j, count = 0, sem_solucao = 0;
	double melhor_residuo = 2e31, criterio_parada, soma_nulo = 0;
	double A1, A2, A3, C1, C2;
```

```
	if( qnt < 5 )
		qnt = 5;

	C1 = 1;
	C2 = 1;

	puts( "\tPARTICLE SWARM OPTIMIZATION ");
	printf( "%d partícula(s) criada(s)\n", qnt );
	puts( "Critério de parada:" );
	scanf( "%lf", &criterio_parada );

	// Melhor posição do bando, resíduo_melhor_posição_bando, melhor posição da partícula, posição da partícula, velocidade da partícula
	Matriz mb, rMB, mp[qnt], s[qnt], v[qnt], rMP, rSi;

	mb = Matriz_init( 1, 1 ); // só iniciando
	// iniciando as partículas com posições e velocidades aleatórias
	for( i = 0; i < qnt; i++ )
	{
		s[i] = Matriz_initAleatoria( A.colunas, 1, max_aleatorio );
		v[i] = Matriz_initAleatoria( A.colunas, 1, max_aleatorio );
		mp[i] = Matriz_copia( s[i] );

		rMP = Matriz_residuo( A, mp[i], b ); // resíduo da melhor posição da partícula

		if( Matriz_normaFrobenius( rMP ) < melhor_residuo )
		{
			Matriz_free( mb );
			melhor_residuo = Matriz_normaFrobenius( rMP );
			mb = Matriz_copia( mp[i] );
			Matriz_free( rMP );
		}

	}

	// Enquanto não alcançar o critério de parada
	while( melhor_residuo > criterio_parada /*&& count < A.colunas * 10000 */)
	{
		// para cada partícula, geraremos escalares aleatórios variando entre 0 e 1 para controlar melhor a velocidade
		for( i = 0; i < qnt; i++ )
		{
			A1 = fmod( rand(), 10e6 ) / 10e6 ;
			A2 = fmod( rand(), 10e6 ) / 10e6 ;
			A3 = fmod( rand(), 10e6 ) / 10e6 ;

			// Aqui garantimos que a velocidade nunca zere
			if( criterio_parada/10 > Matriz_normaFrobenius( v[i] ) )
			{
				Matriz_free( v[i] );
				v[i] = Matriz_initAleatoria( s[i].linhas, 1, max_aleatorio );
```

```
                        }

                        // Ajustamos a velocidade
                        for( j = 0; j < v[i].linhas; j++ )
                                v[i].valor[j][0] = ( A1* v[i].valor[j][0] ) + ( A2 * C1 * ( mp[i].valor[j]
[0] - s[i].valor[j][0] ) ) + ( A3 * C2 * ( mb.valor[j][0] - s[i].valor[j][0] ) );
                        // Ajustamos a posição
                        for( j = 0; j < s[i].linhas; j++ )
                                s[i].valor[j][0] += v[i].valor[j][0];

                        // Verificamos a posição da partícula, para saber se é a melhor posição desta
partícula
                        rSi = Matriz_residuo( A, s[i], b ); // residuo da posição da partícula
                        rMP = Matriz_residuo( A, mp[i], b ); // resíduo da melhor posição da partícula

                        if( Matriz_normaFrobenius( rSi ) < Matriz_normaFrobenius( rMP ) )
                        {
                                Matriz_free( mp[i] );
                                mp[i] = Matriz_copia( s[i] );
                        }

                        // Verificamos se a melhor posição da partícula é a melhor posição do bando
                        rMB = Matriz_residuo( A, mb, b ); // resíduo da melhor posição do bando

                        if( Matriz_normaFrobenius( rMP ) < Matriz_normaFrobenius( rMB ) )
                        {
                                Matriz_free( mb );
                                mb = Matriz_copia( mp[i] );

                                Matriz_free( rMB );
                                rMB = Matriz_residuo( A, mb, b );
                                melhor_residuo = Matriz_normaFrobenius( rMB );

                                Matriz_free( rMP );

                                sem_solucao = 0;
                        }
                        count++;
                }

                sem_solucao++;

                // Se após 300 * qnt iterações não houver melhora na posição do bando de partículas,
assumiremos que o sistema não tem solução
                if( sem_solucao >= 300 )
                {
                        printf( "O sistema não tem solução. Exibindo a solução para a melhor norma
do resíduo\n" );
                        break;
                }
```

```
	}

	printf( "%d iterações feitas\n", count );
	printf( "Norma residual da solução = %.20lf\n", melhor_residuo );
	puts( "Vetor solução encontrado" );
	for( i = 0; i < mb.linhas; i++ )
	{
		printf("%.3lf\n", mb.valor[i][0] );
	}

	//Liberar espaço alocado
	for( i = 0; i < qnt; i++ )
	{
		Matriz_free( s[i] );
		Matriz_free( v[i] );//exemplo
		Matriz_free( mp[i] );
	}

	Matriz_free( rMB );

	return mb;
}

int main( void )
{
	srand( time( NULL ) );

	FILE  *entrada, *saida;

	entrada = fopen( "entrada.txt", "r" );
	saida = fopen( "saida.txt", "w" );

	puts( "\tEXERCÍCIO 3" );
	int linhas, colunas, i, j;
	fscanf( entrada, "%d %d", &linhas, &colunas );

	puts( "Limite superior aleatório para criação dos vetores:" );
	double max_aleatorio;
	scanf( "%lf", &max_aleatorio );

	Matriz A = Matriz_init( linhas, colunas);
	Matriz b = Matriz_init( linhas, 1 );

	Matriz_fscanf( entrada, A );
	Matriz_fscanf( entrada, b );

	puts( "Matriz A (lida no arquivo)" );
	Matriz_fprintf( stdout, A, '\n' );

	puts("\nVetor b (lido no arquivo)" );
	Matriz_fprintf( stdout, b, '\n' );
```

```
	//A = Matriz_transp( A, 0 );

	// xPSO guardará a solução x pelo PSO
	Matriz xPSO = PSO( A, b, max_aleatorio );

	Matriz_fprintf( saida, A, '\n' );
	Matriz_fprintf( saida, b, '\n' );
	Matriz_fprintf( saida, xPSO, '\n' );

	Matriz_free( A );
	Matriz_free( b );
	Matriz_free( xPSO );
}
```

matrizlib.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "matrizlib.h"

static int i, j, k;
short int Matriz_debug = 0;

//aloca espaco para uma matriz NxM
Matriz Matriz_init (unsigned int linhas, unsigned int colunas) {

	Matriz matriz;

	matriz.linhas = linhas;
	matriz.colunas = colunas;
	matriz.triangular = NAO;

	if ( !( matriz.valor = (double**)malloc(linhas*sizeof(double*)) ) ) {
		if (Matriz_debug == 1) printf ("Nao foi possivel criar matriz\n");
	}
	for (i = 0; i < linhas; i++) {
		if ( !( matriz.valor[i] = (double*)malloc(colunas*sizeof(double) ) ) ) {
			if (Matriz_debug == 1) printf ("Nao foi possivel criar matriz\n");
		}
	}

	return matriz;

}


//aloca espaco para uma matriz triangular
Matriz Matriz_initTriang (unsigned int linhas, orientacaoTriang orientacao) {

	Matriz matriz;
```

```
	matriz.linhas = linhas;
	matriz.colunas = linhas;
	matriz.triangular = orientacao;

	if ( !( matriz.valor = (double**)malloc(linhas*sizeof(double) ) ) ) {
		if (Matriz_debug == 1) printf ("Erro: nao foi possivel criar matriz\n");
	}
	for (i = 0; i < linhas; i++) {
		if (orientacao == INFERIOR) {
			if ( !( matriz.valor[i] = (double*)malloc((i+1)*sizeof(double) ) ) ) {
				if (Matriz_debug == 1) printf ("Erro: nao foi possivel criar matriz\n");
			}
		} else if (orientacao == SUPERIOR) {
			if ( !( matriz.valor[i] = (double*)malloc((linhas-i)*sizeof(double) ) ) ) {
				if (Matriz_debug == 1) printf ("Erro: nao foi possivel criar matriz\n");
			}
		}
	}

	return matriz;

}

//libera espaco alocado para uma matriz de ordem > 1x1, indepfimente de seu tipo
void Matriz_free (Matriz matriz) {

	for (i = 0; i < matriz.linhas; i++) {
		free(matriz.valor[i]);
		matriz.valor[i] = NULL;
	}
	free(matriz.valor);
	matriz.valor = NULL;

}

//cria uma copia de uma matriz
Matriz Matriz_copia (Matriz matriz) {
	Matriz copia;
	unsigned int parada;

	//cria matriz de acordo com seu tipo
	if (matriz.triangular == NAO) copia = Matriz_init(matriz.linhas, matriz.colunas);
	else copia = Matriz_initTriang(matriz.linhas, matriz.triangular);

	//copia os valores da matriz para a copia
	for (i = 0; i < matriz.linhas; i++) {
		j = 0;

		if (matriz.triangular == INFERIOR) {
			while (j <= i) {
				copia.valor[i][j] = matriz.valor[i][j];
				j++;
```

```
                }
        }else if (matriz.triangular == SUPERIOR) {
                while (j < matriz.colunas - i) {
                        copia.valor[i][j] = matriz.valor[i][j];
                        j++;
                }
        }else{
                while (j < matriz.colunas) {
                        copia.valor[i][j] = matriz.valor[i][j];
                        j++;
                }
        }

    }

    return copia;

}


//transfere os valores de A para B
void Matriz_transf (Matriz A, Matriz B) {

    if (Matriz_debug == 1) {
        if (A.linhas != B.linhas || A.colunas != B.colunas) {
            printf ("Aviso: transferindo valores entre matrizes de dimensoes
diferentes\n");
        }
    }

    for (i = 0; i < A.linhas; i++) {
        for (j = 0; j < A.colunas; j++) {
            B.valor[i][j] = A.valor[i][j];
        }
    }

}


//retorna a transposta de uma matriz
Matriz Matriz_transp (Matriz matriz, short int copyFlag) {

    Matriz transp;

    //transposta de matriz nao triangular
    if (matriz.triangular == NAO) {
        transp = Matriz_init(matriz.colunas, matriz.linhas);
        for (i = 0; i < matriz.linhas; i++)
            for (j = 0; j < matriz.colunas; j++)
                transp.valor[j][i] = matriz.valor[i][j];
    }
    //transposta de matriz triangular superior
```

```
	else if (matriz.triangular == SUPERIOR) {
		transp = Matriz_initTriang(matriz.linhas, INFERIOR);
		for (i = 0; i < matriz.linhas; i++)
			for (j = 0; j < matriz.colunas-i; j++)
				transp.valor[j+i][i] = matriz.valor[i][j];
	}
	//transposta de matriz triangular inferior
	else {
		transp = Matriz_initTriang(matriz.linhas, SUPERIOR);
		for (j = 0; j < matriz.colunas; j++)
			for (i = j; i < matriz.linhas; i++)
				transp.valor[j][i-j] = matriz.valor[i][j];
	}

	if (copyFlag == 1) {
		Matriz_free(matriz);
		matriz = transp;
	}

	return transp;

}

//cria vetor a partir de linha de uma matriz
Matriz Matriz_vetorLinha (Matriz matriz, int linha) {
	Matriz vetor = Matriz_init (matriz.colunas, 1);

	for (k = 0; k < matriz.colunas; k++)
		vetor.valor[k][0] = matriz.valor[linha][k];

	return vetor;
}

//cria vetor a partir de coluna de uma matriz
Matriz Matriz_vetorColuna (Matriz matriz, int coluna) {
	Matriz vetor = Matriz_init (matriz.linhas, 1);

	for (k = 0; k < matriz.linhas; k++)
		vetor.valor[k][0] = matriz.valor[k][coluna];

	return vetor;
}

//recebe uma matriz com dimensoes ja definidas da entrada selecionada
void Matriz_fscanf (FILE *entrada, Matriz matriz) {

	for (i = 0; i < matriz.linhas; i++)
		for (j = 0; j < matriz.colunas; j++)
			fscanf (entrada, "%lf", &matriz.valor[i][j]);

}
```

```
//cria string de um numero decimal sem 0s a direita
char* stringLimpa(double valor) {
	static char string[100];

	//transforma o numero em uma string
	sprintf (string, "%lf", valor);

	//posiciona k no final da string
	for (k = 0; string[k+1] != '\0'; k++);

	//itera a string a partir do seu ultimo caractere, adicionando o fim de string no lugar dos '0's
	while (string[k] == '0') {
		string[k] = '\0';
		k--;
	}
	//caso ao final do loop anterior string[k] for '.' significa que temos um numero inteiro, entao adicionamos um fim de string no lugar do ponto
	if (string[k] == '.') string[k] = '\0';

	//caso a imprecisao da conversao binario-decimal resulte em -0 transforma-o em 0
	if (string[0] == '-' && string[1] == '0' && string[2] == '\0') {
		string[0] = '0';
		string[1] = '\0';
	}

	//retorna a string do numero sem 0s a direita
	return string;

}

//Imprime matriz na saida selecionada
void Matriz_fprintf (FILE *saida, Matriz matriz, char ultimo) {

	//se a matriz nao eh triangular
	if (matriz.triangular == NAO) {
		for (i = 0; i < matriz.linhas; i++) {
			for (j = 0; j < matriz.colunas; j++) {
				fprintf (saida, "%s ", stringLimpa(matriz.valor[i][j]));
			}
			fprintf (saida, "\n");
		}
	}

	//se a matriz eh triangular superior
	else if (matriz.triangular == SUPERIOR) {
		for (i = 0; i < matriz.linhas; i++) {
			for (j = 0; j < matriz.colunas; j++) {

				if (j >= i) {
					fprintf (saida, "%s ", stringLimpa(matriz.valor[i][j-i]));
				}else {
```

```
                                        fprintf (saida, "0 ");
                                    }

                        }
                        fprintf (saida, "\n");
                }
        }

        //se a matriz eh triangular inferior
        else {
                for (i = 0; i < matriz.linhas; i++) {
                        for (j = 0; j < matriz.colunas; j++) {

                                    if (j > i) {
                                        fprintf (saida, "0 ");
                                    }else{
                                        fprintf (saida, "%s ", stringLimpa(matriz.valor[i][j]));
                                    }

                        }
                        fprintf (saida, "\n");
                }
        }

        fprintf (saida, "%c", ultimo);

}


//efetua soma entre duas matrizes
Matriz Matriz_soma (Matriz A, Matriz B, short int copyFlag) {

        //verificacao de possibilidade da operacao
        if (A.linhas != B.linhas || A.colunas != B.colunas) {
                if (Matriz_debug == 1) printf ("Erro: operacao com dimensoes incompativeis\n");
        }

        Matriz result;

        if (copyFlag == 0)
                result = Matriz_init(A.linhas, A.colunas);
        else if (copyFlag == 1)
                result = A;
        else
                result = B;

        for (i = 0; i < A.linhas; i++) {
                for (j = 0; j < A.colunas; j++) {
                        result.valor[i][j] = A.valor[i][j]+B.valor[i][j];
                }
        }
```

```
	return result;

}


//efetua subtracao entre duas matrizes
Matriz Matriz_sub (Matriz A, Matriz B, short int copyFlag) {

	//verificacao de possibilidade da operacao
	if (A.linhas != B.linhas || A.colunas != B.colunas) {
		if (Matriz_debug == 1) printf ("Erro: operacao com dimensoes incompativeis\n");
	}

	Matriz result;

	if (copyFlag == 0)
		result = Matriz_init(A.linhas, A.colunas);
	else if (copyFlag == 1)
		result = A;
	else
		result = B;

	for (i = 0; i < A.linhas; i++) {
		for (j = 0; j < A.colunas; j++) {
			result.valor[i][j] = A.valor[i][j]-B.valor[i][j];
		}
	}

	return result;

}

//somatorio dos produtos A[i][j]*B[j][i] com matrizes nao triangulares
static double Somatorio_MatrizMult(double* linhaA, double** B, int colunaB, int limit) {

	double soma = 0;

	for (k = 0; k < limit; k++) {
		soma += linhaA[k]*B[k][colunaB];
	}

	return soma;

}

//somatorio dos produtos A[i][j]*B[j][i] com matrizes triangulares
static double Somatorio_MatrizMultTriang(double* linhaA, double** B, int colunaB, int limit) {

	double soma = 0;

	for (k = 0; k <= limit; k++) {
		soma += linhaA[k]*B[k][colunaB-k];
```

```
	}

	return soma;

}


//multiplicacao entre duas matrizes
Matriz Matriz_mult (Matriz A, Matriz B, short int copyFlag) {

	//verificacao de possibilidade da operacao
	if (Matriz_debug == 1) {
		if (A.linhas != B.colunas || A.colunas != B.linhas) {
			printf ("Erro: operacao com dimensoes incompativeis\n");
		}
		if (A.triangular == INFERIOR && B.triangular != SUPERIOR) {
			printf ("Erro: operacao com esses tipos de matrizes nao eh suportado\n");
		}
	}

	Matriz result;

	result = Matriz_init(A.linhas, B.colunas);

	//operacao entre matrizes nao triangulares
	if (A.triangular == NAO && B.triangular == NAO) {
		for (i = 0; i < A.linhas; i++) {
			for (j = 0; j < B.colunas; j++) {
				result.valor[i][j] = Somatorio_MatrizMult( A.valor[i], B.valor, j,
A.colunas);
			}
		}
	}

	//operacao com A triangular inferior e B triangular superior
	if (A.triangular == INFERIOR && B.triangular == SUPERIOR) {

		for (i = 0; i < A.linhas; i++) {
			for (j = 0; j < B.colunas; j++) {
				result.valor[i][j] = Somatorio_MatrizMultTriang( A.valor[i], B.valor,
j, i);
			}
		}

	}

	if (copyFlag == 1) {
		Matriz_free(A);
		A = result;
	}else if (copyFlag == 2) {
		Matriz_free(B);
		B = result;
```

```
	}

	return result;

}


//modulo
double mod (double numero) {
	if (numero < 0)
		numero *= -1;

	return numero;
}

//multiplicacao de matriz por escalar
Matriz Matriz_multEscalar (double escalar, Matriz matriz, short int copyFlag) {

	Matriz result;

	if (copyFlag == 0)
		result = Matriz_init(matriz.linhas, matriz.colunas);
	else
		result = matriz;

	for (i = 0; i < matriz.linhas; i++) {
		for (j = 0; j < matriz.colunas; j++) {
			result.valor[i][j] = escalar * matriz.valor[i][j];
		}
	}

	return result;

}


//substituicao para tras
Matriz Matriz_substTras(Matriz A, Matriz b, short int copyFlag) {

	//verificacao de erro
	if (Matriz_debug == 1) {
		if (b.colunas > 1) printf ("Aviso: b nao eh um vetor\n");
		if (A.linhas != b.linhas) printf ("Erro: dimensoes incompativeis entre A e b\n");
	}

	Matriz vetor;
	double soma;

	if (copyFlag == 0)
		vetor = Matriz_init(A.linhas, 1);
	else
		vetor = b;
```

```
	for (i = A.linhas-1; i >= 0; i--) {

		for (j = A.colunas-i-1, soma = 0; j > 0; j--) {
			soma += A.valor[i][j]*vetor.valor[j+i][0];
		}

		vetor.valor[i][0] = (b.valor[i][0] - soma) / A.valor[i][0];

	}

	return vetor;

}

//substituicao para frente
Matriz Matriz_substFrente (Matriz A, Matriz b, short int copyFlag) {

	//verificacao de erro
	if (Matriz_debug == 1) {
		if (b.colunas > 1) printf ("Aviso: b nao eh um vetor\n");
		if (A.linhas != b.linhas) printf ("Erro: dimensoes incompativeis entre A e b\n");
	}

	Matriz vetor;
	double soma;

	if (copyFlag == 0)
		vetor = Matriz_init(A.linhas, 1);
	else
		vetor = b;

	for (i = 0; i < A.linhas; i++) {
		for (j = 0, soma = 0; j < i; j++) {
			soma += A.valor[i][j]*vetor.valor[j][0];
		}

		vetor.valor[i][0] = (b.valor[i][0] - soma) / A.valor[i][i];

	}

	return vetor;

}

//pivo de uma matriz
static void pivo (Matriz matriz, int ref) {
	double* temp;
	int maior;

	for (k = ref ; k < matriz.linhas-1; k++) {
		maior = k;
```

```
		for (j = ref+1; j < matriz.linhas; j++) {
			if (matriz.valor[maior][ref] < matriz.valor[j][ref]) {
				maior = j;
			}
		}
		if (maior != k) {
			temp = matriz.valor[k];
			matriz.valor[k] = matriz.valor[maior];
			matriz.valor[maior] = temp;
		}
	}
}

//pivo de duas matrizes
static void pivo2 (Matriz A, Matriz B, int ref) {
	double* temp;
	int maior;

	for (k = ref ; k < A.linhas-1; k++) {
		maior = k;
		for (j = ref+1; j < A.linhas; j++) {
			if (A.valor[maior][ref] < A.valor[j][ref]) {
				maior = j;
			}
		}
		if (maior != k) {
			temp = A.valor[k];
			A.valor[k] = A.valor[maior];
			A.valor[maior] = temp;
			temp = B.valor[k];
			B.valor[k] = B.valor[maior];
			B.valor[maior] = temp;
		}
	}
}

//aplica eliminacao gaussiana a uma matriz
Matriz Matriz_eliminacaoGauss (Matriz matriz, short int copyFlag) {

	if (matriz.linhas != matriz.colunas)
		if (Matriz_debug == 1) printf ("Erro: aplicando Eliminacao Gaussiana em uma
matriz nao quadrada\n");

	double f;
	Matriz result;

	if (copyFlag == 0)
		result = Matriz_copia(matriz);
	else
		result = matriz;

	//eliminacao com pivot parcial, gera uma matriz triangular superior
```

```
	for (i = 0; i < result.linhas-1; i++) {
		pivo(result, i);

		for (j = i+1; j < result.linhas; j++) {
				f = result.valor[j][i] / result.valor[i][i];
				for (k = 0; k < result.colunas; k++)
					result.valor[j][k] = result.valor[j][k] - f*result.valor[i][k];
		}
	}

	return result;

}

//aplica eliminacao gaussiana em A e B, onde B sofre as mesmas transformacoes que A
void Matriz_eliminacaoGauss2 (Matriz A, Matriz B) {

	if (Matriz_debug == 1) {
		if (A.linhas != A.colunas) printf ("Erro: aplicando Eliminacao Gaussiana em uma
matriz nao quadrada\n");
		if (A.linhas != B.linhas) printf ("Erro: aplicando Eliminacao Gaussiana entre
matrizes de dimensoes incompativeis\n");
	}

	double f;

	//eliminacao com pivot parcial
	for (i = 0; i < A.linhas-1; i++) {
		pivo2(A, B, i);

		for (j = i+1; j < A.linhas; j++) {
				f = A.valor[j][i] / A.valor[i][i];
				for (k = 0; k < A.colunas; k++)
					A.valor[j][k] -= f*A.valor[i][k];
				for (k = 0; k < B.colunas; k++)
					B.valor[j][k] -= f*B.valor[i][k];
		}
	}

}

//determinante de uma matriz
double Matriz_det(Matriz matriz) {

	if (matriz.linhas != matriz.colunas)
		if (Matriz_debug == 1) printf ("Erro: calculando determinante de uma matriz nao
quadrada\n");

	Matriz reduzida = Matriz_eliminacaoGauss(matriz, 0);
	double det = 1;

	for (i = 0; i < reduzida.linhas && det != 0; i++)
```

```
            det *= reduzida.valor[i][i];

        Matriz_free(reduzida);
        return det;

}


//inversa de uma matriz
Matriz Matriz_inversa(Matriz matriz, short int copyFlag) {

        if (matriz.linhas != matriz.colunas)
                if (Matriz_debug == 1) printf ("Erro: calculando inversa de matriz nao quadrada\n");

        Matriz inversa, copia;
        double f;

        //cria uma copia de matriz
        copia = Matriz_copia(matriz);

        //inicializacao da matriz identidade
        if (copyFlag == 0)
                inversa = Matriz_init(matriz.linhas, matriz.colunas);
        else
                inversa = matriz;

        for (i = 0; i < inversa.linhas; i++) {
                for (j = 0; j < inversa.colunas; j++) {
                        if (i != j)
                                inversa.valor[i][j] = 0;
                        else
                                inversa.valor[i][j] = 1;
                }
        }

        //gera matriz triangular superior
        Matriz_eliminacaoGauss2 (copia, inversa);

        //zera os elementos fora da diagonal principal
        for (i = 0; i < copia.linhas-1; i++) {
                for (j = i+1; j < copia.colunas; j++) {
                        if (copia.valor[i][j] != 0) {
                                        f = copia.valor[i][j] / copia.valor[j][j];
                                        for (k = 0; k < copia.colunas; k++) {
                                                copia.valor[i][k] -= f*copia.valor[j][k];
                                                inversa.valor[i][k] -= f*inversa.valor[j][k];
                                        }
                        }
                }
        }

        //transforma os elementos da diagonal principal em 1
```

```
	for (i = 0; i < copia.linhas; i++) {
		if (copia.valor[i][i] != 0) {
			f = 1/copia.valor[i][i];
			copia.valor[i][i] = 1;
			for (j = 0; j < inversa.colunas; j++)
				inversa.valor[i][j] *= f;
		}else{
			printf ("Erro: matriz nao eh inversivel\n");
			Matriz_free(inversa);
			break;
		}
	}

	Matriz_free(copia);
	return inversa;

}

//somatorio com lei de formacao i^2
static double Somatorio_powi2 (double *num, int inicio, int fim) {

	double soma = 0;

	for ( ; inicio < fim; inicio++)
		soma += pow(num[inicio], 2);

	return soma;

}

//somatorio com lei de formacao matriz[j][n]*matriz[i][n]
static double Somatorio_mult (double **matriz, int inicio, int linha, int coluna) {

	double soma = 0;

	for ( ; inicio < coluna; inicio++)
		soma += matriz[coluna][inicio]*matriz[linha][inicio];

	return soma;

}


//retorna o fator de Cholesky de uma matriz se possivel
Matriz Matriz_fatorCholesky (Matriz matriz) {

	Matriz fator = Matriz_initTriang(matriz.linhas, INFERIOR);
	double temp;

	for (i = 0; i < fator.linhas; i++) {

		//elementos fora da diagonal principal
```

```
                for (j = 0; j < i; j++) {
                        fator.valor[i][j] = ( matriz.valor[i][j] - Somatorio_mult(fator.valor, 0, i, j) ) /
fator.valor[j][j];
                }

                //elementos da diagonal principal, apos o loop anterior j == i
                temp = matriz.valor[i][i] - Somatorio_powi2(fator.valor[i], 0, i);
                if (temp > 0 || temp == 0 && i == fator.linhas-1) {
                        fator.valor[i][i] = sqrt(temp);
                }else{
                        if (Matriz_debug == 1) printf ("Erro: Matriz nao possui fator de
Cholesky\n");
                        Matriz_free(fator);
                        break;
                }

        }

        return fator;

}


//verifica se a matriz eh uma matriz de vandermonde, 1 = verdadeiro; 0 = falso
short int Matriz_ehVandermonde (Matriz matriz) {

        double x;

        //verifica se a matriz eh quadrada
        if (matriz.linhas != matriz.colunas) return 0;

        //verifica se algum elemento da primeira coluna eh diferente de um
        for (i = 0; i < matriz.linhas; i++)
                if (matriz.valor[i][0] != 1) return 0;

        //verifica se os elementos depois da primeira coluna seguem a regra de formacao
        for (i = 0; i < matriz.linhas; i++) {
                x = matriz.valor[i][1];
                for (j = 2; j < matriz.colunas; j++) {
                        if (matriz.valor[i][j] != pow(x, j)) return 0;
                }
        }

        return 1;
}

//norma infinito para vetor
double Matriz_normaInfinito (Matriz vetor) {
        if (Matriz_debug == 1) {
                if (vetor.colunas > 1) printf ("Aviso: Calculando norma infinito de nao-vetor\n");
        }
```

```
	double maior = mod(vetor.valor[0][0]);

	for (i = 0; i < vetor.linhas; i++)
		if (mod(vetor.valor[i][0]) > maior)
			maior = mod(vetor.valor[i][0]);

	return maior;
}

//norma de frobenius para matriz
double Matriz_normaFrobenius (Matriz matriz) {

	double somatorio = 0;

	for (i = 0; i < matriz.linhas; i++)
		for (j = 0; j < matriz.colunas; j++)
			somatorio += matriz.valor[i][j]*matriz.valor[i][j];

	return sqrt(somatorio);

}


//norma linha para matriz
double Matriz_normaLinha (Matriz matriz) {

	double maior = 0,
		soma = 0;

	for (i = 0; i < matriz.linhas; i++, soma = 0) {
		for (j = 0; j < matriz.colunas; j++) {
			soma += mod(matriz.valor[i][j]);
		}

		if (soma > maior)
			maior = soma;
	}

	return maior;

}

//norma coluna para matriz
double Matriz_normaColuna (Matriz matriz) {

	double maior = 0;
	double soma = 0;

	for (j = 0; j < matriz.colunas; j++, soma = 0) {
		for (i = 0; i < matriz.linhas; i++) {
			soma += mod(matriz.valor[i][j]);
		}
```

```
                if (soma > maior)
                        maior = soma;
        }

        return maior;
}


//produto interno
double Matriz_produtoInterno (Matriz a, Matriz b) {
        if (Matriz_debug == 1) {
                if (a.colunas > 1 || b.colunas > 1) printf ("Aviso: Calculando produto interno de nao-
vetores\n");
                if (a.linhas != b.linhas) printf ("Erro: Calculando produto interno de vetores de
dimensoes diferentes\n");
        }

        double produto = 0;

        for (i = 0; i < a.linhas; i++)
                produto += a.valor[i][0]*b.valor[i][0];

        return produto;

}


//angulo entre dois vetores
double Matriz_anguloVetores (Matriz a, Matriz b) {
        if (Matriz_debug == 1) {
                if (a.colunas > 1 || b.colunas > 1) printf ("Aviso: Calculando angulo de nao-
vetores\n");
                if (a.linhas != b.linhas) printf ("Erro: Calculando angulo entre vetores de dimensoes
diferentes\n");
        }

        double cos;

        cos = Matriz_produtoInterno(a, b)/(Matriz_normaFrobenius(a)*Matriz_normaFrobenius(b));

        return cos;

}


//determinante de matriz de vandermonde
double Matriz_detVandermonde (Matriz matriz) {
        double det = 1;

        for (j = 0; j < matriz.linhas && det != 0; j++) {
                for (i = j+1; i < matriz.linhas && det != 0; i++) {
```

```
                det *= matriz.valor[i][1] - matriz.valor[j][1];
            }
        }

        return det;
}


//resolve o sistema R*Rt*x = b, onde R eh o fator Cholesky (triangular inferior) de uma matriz e Rt sua transposta
Matriz Matriz_solucaoCholesky (Matriz R, Matriz b, short int copyFlag) {

        if (Matriz_debug == 1) {
            if (R.triangular != INFERIOR) printf ("Erro: R nao eh triangular inferior\n");
            if (b.colunas > 1) printf ("Erro: b nao eh um vetor\n");
        }

        Matriz x, y, Rt;

        //chamo Rt*x de y, resolvo Ry = b por substituicao para frente e obtenho y
        y = Matriz_substFrente (R, b, copyFlag);
        //resolvo Rt*x = Y por substituicao para tras e obtenho x
        Rt = Matriz_transp(R, 1);
        x = Matriz_substTras (Rt, y, copyFlag);


        //limpa memoria utilizada na resolucao do problema
        Matriz_free(y);

        return x;

}


//verifica se a matriz satisfaz o criterio das linhas
short int Matriz_criterioLinhas (Matriz matriz) {

        double result;

        for (i = 0; i < matriz.linhas; i++) {

            result = 0;

            for (j = 0; j < matriz.colunas; j++) {
                //se j = i, pulamos essa coluna
                if (j == i) continue;

                result += mod(matriz.valor[i][j]);

            }

            result /= mod(matriz.valor[i][i]);
```

```
                //result viola o criterio das linhas
                if (result > 1) return 0;

        }

        //nao existe nenhum result que viole o criterio das linhas
        return 1;

}

//verifica se a matriz satisfaz o criterio das colunas
short int Matriz_criterioColunas (Matriz matriz) {

        double result;

        for (j = 0; j < matriz.colunas; j++) {

                result = 0;

                for (i = 0; i < matriz.linhas; i++) {
                        //se j = i, pulamos essa linha
                        if (j == i) continue;

                        result += mod(matriz.valor[i][j]);

                }

                result /= mod(matriz.valor[j][j]);

                //result viola o criterio das colunas
                if (result > 1) return 0;

        }

        //nao existe nenhum result que viole o criterio das colunas
        return 1;

}


//verifica se a matriz satisfaz o criterio de Sassenfeld
short int Matriz_criterioSassenfeld (Matriz matriz) {

        double result,
                B[matriz.linhas];

        //inicia todos os elementos de B como sendo 1
        for (i = 0; i < matriz.linhas; i++)
                B[i] = 1;

        for (i = 0; i < matriz.linhas; i++) {
```

```
		result = 0;

		for (j = 0; j < matriz.colunas; j++) {
			//se j = i, pulamos essa coluna
			if (j == i) continue;

			result += mod(matriz.valor[i][j]) * B[j];
		}

		B[i] = result/mod(matriz.valor[i][i]);

		//B[i] viola o criterio de sassenfeld
		if (B[i] > 1) {
			free(B);
			return 0;
		}

	}

	//nao existe nenhum B[i] que viole o criterio de Sassenfeld
	return 1;

}


//verifica se a matriz eh uma matriz banda, 1 = verdadeiro; 0 = falso
short int Matriz_ehBanda (Matriz matriz) {

	int diagNulasInf = 0,
	   diagNulasSup = 0;

	if (matriz.linhas != matriz.colunas) {
		return 0;
	}

	//verifica diagonais inferiores da matriz
	for (i = matriz.linhas - 1; i > 0; i--) {
		for (j = 0, k = i; k < matriz.linhas; j++, k++) {
			if (matriz.valor[k][j] != 0) {
				goto verifica_sup;
			}
		}
		diagNulasInf++;
	}

	//verifica diagonais superiores da matriz
	verifica_sup:

	for (j = matriz.colunas-1; j > 0; j--) {
		for (i = 0, k = j; k < matriz.colunas; i++, k++) {
			if (matriz.valor[i][k] != 0) {
```

```
                        goto retorno;
                    }
                }
                diagNulasSup++;
        }

        //verifica se a matriz eh uma matriz banda com base no numero de diagonais nulas
        retorno:

        if (diagNulasSup == diagNulasInf && diagNulasSup > 0)  return 1;
        else return 0;

}

//verifica igualdade entre duas matrizes
short int Matriz_iguais (Matriz A, Matriz B, double tol) {
        if (A.linhas != B.linhas || A.colunas != B.colunas) return 0;

        Matriz r;
        double dif;

        r = Matriz_sub(A, B, 0);
        dif  = Matriz_normaFrobenius(r);
        Matriz_free(r);

        if (dif > tol) return 0;

        return 1;
}

// Forma uma matriz a partir de um intervalo [inicio, fim) de vetores nx1
Matriz Matriz_vetorParaMatriz( Matriz* vetores, int inicio, int fim ) {
        Matriz matriz = Matriz_init( vetores[0].linhas, fim - inicio );

        for( i = inicio; i < fim; i++ )
                for( j = 0; j < vetores[0].linhas; j++ )
                        matriz.valor[j][i - inicio] = vetores[i].valor[j][0];

        return matriz;

}
```

Questão 5 – LI

LI-FaltaB.c

```
#include <stdio.h>
#include <math.h>

void check_Linear_Dependency(int n, int dim, double *matrix);
void ordering_Matrix(int n, int dim, double *matrix);
void calculate_Unitary_Vector(int n, int dim, double *matrix);
double CalculoDeterminante(int ordem , double *matriz);
double cofactor(int ordem, double *matriz, int linha, int coluna);

int main(void)
{

	printf("-----------VETORES LINEARMENTE INDEPENDENTES-----------\n\n");

	int n = 0;
	int dim = 0;
	int forcar_entrada_correta = 0;

	while (n < 1 || dim < 1)
	{
		if (forcar_entrada_correta > 0)
		{
			printf("\nEntre com valores iguais a 1 ou maiores!\n");
		}

		printf("Entre com o número de vetores a serem analizados\n");
		scanf("%d", &n);

		printf("Entre com a dimensão dos vetores a serem analizados\n");
		scanf("%d", &dim);

		forcar_entrada_correta ++;
	}

	double matriz[n][dim];


	printf("Entre com %d vetores de %d dimensões, não nulos\n", n, dim);

	for (int i = 0; i < n; i ++)
	{

		for (int j = 0; j < dim; j ++)
		{
			scanf("%lf", &matriz[i][j]);
		}
```

```
	}



	if (n == dim && CalculoDeterminante(n, &matriz[0][0]) == 0)
	{
		printf("\n\nLinearmente Dependente! (Det = 0)\n");
		calculate_Unitary_Vector(n, dim, &matriz[0][0]);
		return 0;
	}

	calculate_Unitary_Vector(n, dim, &matriz[0][0]);

	check_Linear_Dependency(n, dim, &matriz[0][0]);

	return 0;

}

void calculate_Unitary_Vector(int n, int dim, double *matrix)
{

	long double norm[n];


	for (int i = 0; i < n; i ++)
	{

		norm[i] = 0;

		for (int j = 0; j < dim; j ++)
		{
			norm[i] = norm[i] + *(matrix + (i*dim) + j) * *(matrix + (i*dim) + j);
		}

		norm[i] = sqrt(norm[i]);

		printf("\n\nVetor Unitário %d = {", i + 1);

		for (int l = 0; l < dim; l ++)
		{
			if (l + 1 < dim)
			{
				printf("%Lf , ", *(matrix + (i * dim) + l) /norm[i] );

			}

			else
			{
				printf("%Lf}", *(matrix + (i * dim) + l) /norm[i] );
```

```
                }
            }

            printf("\n\n");
        }

        return;

}

void check_Linear_Dependency(int n, int dim, double *matrix)
{
        int same_horizontal = 0;
        int same_vertical = 0;
        int zeros = 0;
        long double norm[n];

        //Checa zeros
        for (int i = 0; i < n; i ++)
        {
            zeros = 0;

            for (int j = 0; j < dim; j ++)
            {
                if (*(matrix + (i*n) + j) == 0)
                {
                    zeros ++;
                }
            }

            if (zeros == dim)
            {
                printf("Linearmente Dependente! (Vetor Nulo)\n");
                return ;
            }
        }

        //CHeca Posto
        if (n > dim)
        {
            printf("Linearmente Dependente! (Mais Vetores que a Dimensão)\n" );
            return;
        }

        for (int i = 0; i < n; i ++)
        {
            norm[i] = 0;

            for (int j = 0; j < dim; j ++)
            {
                norm[i] = norm[i] + *(matrix + (i*dim) + j) * *(matrix + (i*dim) + j);
            }
```

```
			norm[i] = sqrt(norm[i]);

		}

		//Checka igualdade por linhas
		for (int i = 0; i < n; i ++)
		{
			for (int j = 0; j < n; j ++)
			{
				same_horizontal = 0;

				for (int k = 0; k < dim; k ++)
				{
					if(i == j)
					{
						break;
					}

					//OBSSSS -> FLOAT SÃO APROXIMAÇÔES... diferencas de (-0.001 a 0.001) são consideradas iguais
					if ((*(matrix + (i*dim) + k)/norm[i]) - (*(matrix + (j*dim) + k)/norm[j]) < 0.001 && (*(matrix + (i*dim) + k)/norm[i]) - (*(matrix + (j*dim) + k)/norm[j]) > -0.001)
					{
						same_horizontal ++;

					}

					if (same_horizontal == dim)
					{
						printf("Linearmente Dependente!\n");
						return;
					}
				}
			}
		}


		printf("Linearmente Independente!\n");
		return;
}

double CalculoDeterminante(int ordem , double *matriz)
{
		double determinante = 0;
		int i, j;
		if(ordem == 1)
			determinante = *matriz;
		else{
			for(j = 0; j < ordem; j++){
```

```
                        determinante = determinante + (*(matriz + j) * cofactor(ordem, &(*matriz),
0, j));
                }
        }

        return determinante;
}

double cofactor(int ordem, double *matriz, int linha, int coluna)
{
        int i, j, aux1 = 0, aux2 = 0, ordem2 = ordem - 1;
        double matrizCofactor[ordem2][ordem2];

        for(i = 0; i<ordem; i++)
        {
                for(j = 0; j<ordem; j++)
                {
                        if(i != linha && j != coluna)
                        {
                                matrizCofactor[aux1][aux2] = *(matriz + (i*ordem) + j);
                                aux2++;
                                if(aux2 >= ordem2)
                                {
                                        aux1++;
                                        aux2 = 0;
                                }
                        }
                }
        }
        return pow(-1, linha+coluna) * CalculoDeterminante(ordem2, &matrizCofactor[0][0]);
}
```

LI-B.c

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

int main (void)
{
        int i, j, n, k, count = 0;
        printf("Insira a quantidade de elementos do vetor:\n");
        scanf ("%d", &n);
        float mat[n][n],vet[n][n], /*vetun [n],*/ neg = 0, temp, rest = 0;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                {
                        printf("Insira o elemento a %d\n do vetor %d", j+1, i+1);
                        scanf("%f", &vet[i][j]);
                }
        }
```

```
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                {
                        mat[i][j]=vet[i][j];

                }
        }


        for(i = 0; i < n - 1; i++)
{
   if(vet[i][i] == 0)
   {
      for(k = i; k < n; k++)
      {
         if(vet[k][i] != 0)
         {
            for(j = 0; j < n; j++)
            {
               temp = vet[i][j];
               vet[i][j] = vet[k][j];
               vet[k][j] = temp;
            }
            k = n;
         }
      }
      count++;
   }

   if(vet[i][i] != 0)
   {
      for(k = i + 1; k < n; k++)
      {
         neg = -1.0 * vet[k][i] /  vet[i][i];
         for(j = i; j < n; j++)
         {
            vet[k][j] = vet[k][j] + (neg * vet[i][j]);
         }
      }
   }
}

temp = 1.0;
// Calcula o determinante
rest = count%2;
for(i = 0; i < n; i++)
   temp *= vet[i][i];

printf("\nDeterminante:\n");
if(rest == 0)
```

```
   printf("%f \n", temp);
else
   printf("%f \n", -1.0 * temp);

if (temp == 0)
{
      printf("Esse sistema é linearmente dependente\n");
}
else
{
      printf("Esse sistema é linearmente independente\n");

}
      for (i = 0; i < n; i++)
      {
            for (j = 0; j < n; j++)
            {
                  printf(" %f\n",mat[i][j]);

            }
      }


      // Vendo se é ortogonal

      float ort=1,ort1=0;
int p=0;
while(p<n){
for (i = 0; i < n; i++)
      {

      ort*=mat[i][p];
   if(i+1==n){
      ort1+=ort;
      ort=1;
      p++;
   }
      }
       for (i = 0; i < n; i++)
      {

   ort*=mat[i][p];
   if(i+1==n){
      ort1+=ort;
      ort=1;
      p++;
   }
      }
      }
      printf("\n\n\n\prod escalar: \n%f",ort1);

      float uni[n];
```

```
   for (i = 0; i < n; i++)
         {
               uni[i]=0;
         }

         for (i = 0; i < n; i++)
         {
                  for (j = 0; j < n; j++)
                  {
         uni[i]+=pow(mat[i][j],2);
         uni[i]=sqrt(uni[i]);

      }
   }
 int t=1;
   if(temp!=0)
   {
         if(ort1==0)
         {
               for(i=0;i<n;i++)
               {
                     if(uni[i]==1)
                     {
                        t++;
                        if(t==n)
                        {
                           printf("\n\n\nOs vetores sao ortonormais\n");
                        }
                     }
                     else
                        printf("\n\n\n\n Os vetores nao sao unitarios");
               }
         }
         else
            printf("\n\n\n\nOs vetores nao sao ortogonais\n\n\n");
   }
   else
      printf("\n\n\n\nOs vetores sao linearmente dependentes");



return 0;

}
```