```
#include "mmult.h"

// Multiplies two matrices together and returns the result in a matrix struct
// First, the result matrix struct is made
// Next, the second matrix is transposed to make the multiplication faster
// Lastly, the matrices are multiplied together and the output is saved in the result matri
x struct
struct matrix *mmult(struct matrix *m1, struct matrix *m2)
{
    uint32_t i;
    uint32_t j;
    uint32_t k;
    float total;
    float transpose[m2->rows * m2->columns];
    struct matrix *result = (struct matrix *)malloc(sizeof(struct matrix));

    result->columns = m2->columns;
    result->rows = m1->rows;
    result->matrix = (float *)malloc(m1->rows * m2->columns * sizeof(float));

    // Parallelized with openmp
    // Transposes the second matrix so that memory is quicker
    #pragma omp parallel shared(transpose,m2,i) private(j) num_threads(4)
    {
        #pragma omp for
        for (i = 0; i < m2->columns; i++) {
            for (j = 0; j < m2->rows; j++) {
                transpose[i * m2->rows + j] = m2->matrix[j * m2->columns + i];
            }
        }
    }

    // Checks to make sure that the matrices can be multiplied together
    if (m1->columns != m2->rows) {
        printf("The columns of the 1st matrix must be equal to the rows of the 2nd matrix!\
n");
        return result;
    }

    // Parallelized with openmp
    // Each row in the first matrix goes through each row in the transposed matrix
    // and corresponding elements in the rows are multiplied. They are then totalled
    // and the result is then placed as an element in the result matrix.
    #pragma omp parallel shared(result,m1,m2,i) private(j,k,total) num_threads(4)
    {
        #pragma omp for
        for (i = 0; i < m1->rows; i++) {
            for (j = 0; j < m2->columns; j++) {
                for (k = 0; k < m1->columns; k++) {
                    total += m1->matrix[i * m1->columns + k] * transpose[j * m2->rows + k];
                }
                result->matrix[i * m2->columns + j] = total;
                total = 0;
            }
        }
    }

    return result;
}

// Allocates memory for struct and matrix array in struct
// Assigns the rows and columns in the struct to the rows and columns passed
// Puts randoms numbers in the matrix from -rangemax/2 to +rangemax/2
struct matrix *mgen(uint32_t rows, uint32_t columns)
{
    uint64_t i;
    uint32_t iseed;
    uint32_t rangemax;
```

```
    struct matrix *gen = (struct matrix *)malloc(sizeof(struct matrix));

    gen->rows = rows;
    gen->columns = columns;
    gen->matrix = (float *)malloc(rows * columns * sizeof(float));

    iseed = (uint32_t)time(NULL);
    srand(iseed);
    rangemax = 100;

    // Casted to an int at the end so easier to check validity of matrix multiply
    // since there can be rounding error with floats
    for(i = 0; i < rows * columns; i++) {
        gen->matrix[i] = (int)(((float)rand() / (float)(RAND_MAX)) * rangemax) - (0.5 * ran
gemax);
    }

    return gen;
}

// Prints each element of matrix
// Prints matrix as a 2D matrix
void mprint(struct matrix *m)
{
    uint64_t i;

    // Each element is separated by a comma and space
    // Rows are incased with [] and are separated with a
    // comma and newline.
    printf("[[");
    for (i = 0; i < (m->rows * m->columns); i++) {
        if (((i + 1) % m->columns == 0) && i != (m->rows * m->columns - 1)) {
            printf("%f],\n [",m->matrix[i]);
        } else if (i != (m->rows * m->columns - 1)) {
            printf("%f, ",m->matrix[i]);
        } else {
            printf("%f]];\n\n", m->matrix[i]);
        }
    }

    return;
}

// Creates txt files for Python script to validate matrix multiply
void createtxt(struct matrix *m, char *filename)
{
    uint64_t i;
    FILE *f;
    char *named = "";

    named = strcat(filename,".txt");

    f = fopen(named,"w+");

    // Puts matrix into file with each element separated by a space
    // Each row is separated by a newline
    for (i = 0; i < (m->rows * m->columns); i++) {
        if ((i + 1) % m->columns != 0) {
            fprintf(f,"%f ",m->matrix[i]);
        } else {
            fprintf(f,"%f\n",m->matrix[i]);
        }
    }

    fclose(f);
    printf("Created %s\n",named);
    return;
}
```