

Multi-Module Design Example: Matrix Operations in C

A more advanced multi-module C example incorporating three reusable modules:

1. **Matrix Multiplication:** Handles multiplying two matrices.
2. **Gauss-Jordan Elimination for Matrix Inverse:** Computes the inverse of a square matrix using the Gauss-Jordan method (augmented matrix with identity, row operations to reduce to identity on left).
3. **Linear Regression using Matrices:** Implements simple linear regression (ordinary least squares) for a dataset, using matrix operations. For data points (X, y) , it computes coefficients $\beta = (X^T * X)^{-1} * X^T * y$, where X is the design matrix (with a column of 1s for intercept).

To make them reusable, each module has its own header (**.h**) for declarations and source (**.c**) for implementations. I'll include helper functions (e.g., transpose, print) in a shared **matrix_utils** module for commonality.

Assumptions and Design

- Matrices are represented as `double **` (dynamically allocated 2D arrays) for flexibility.
- Error handling is basic (e.g., check dimensions, invertibility).
- The main program (`main.c`) demonstrates all modules: Multiplies two matrices, inverts one, and performs linear regression on sample data.
- Memory management: Each module allocates/frees as needed; caller frees results.
- For simplicity, no external libraries (e.g., no LAPACK); pure C.
- Fixed max size isn't used; dynamic allocation via `malloc`.

File Structure

- `matrix_utils.h` / `matrix_utils.c`: Shared helpers (create, free, transpose, print matrices).
- `matrix_mult.h` / `matrix_mult.c`: Matrix multiplication.
- `gauss_jordan.h` / `gauss_jordan.c`: Matrix inverse via Gauss-Jordan.
- `linear_regression.h` / `linear_regression.c`: Linear regression using matrices.
- `main.c`: Entry point to test everything.

`matrix_utils.h`

```
#ifndef MATRIX_UTILS_H
#define MATRIX_UTILS_H
```

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

double** create_matrix(int rows, int cols);
void free_matrix(double** mat, int rows);
double** transpose_matrix(double** mat, int rows, int cols, int* new_rows, int*
new_cols);
void print_matrix(double** mat, int rows, int cols);

#endif /* MATRIX_UTILS_H */
```

matrix_utils.c

```
#include "matrix_utils.h"

double** create_matrix(int rows, int cols) {
    double** mat = (double**)malloc(rows * sizeof(double*));
    if (!mat) return NULL;
    for (int i = 0; i < rows; i++) {
        mat[i] = (double*)malloc(cols * sizeof(double));
        if (!mat[i]) return NULL; // Simplified error; in real code, free prior.
    }
    return mat;
}

void free_matrix(double** mat, int rows) {
    if (!mat) return;
    for (int i = 0; i < rows; i++) {
        free(mat[i]);
    }
    free(mat);
}

double** transpose_matrix(double** mat, int rows, int cols, int* new_rows, int*
new_cols) {
    *new_rows = cols;
    *new_cols = rows;
    double** trans = create_matrix(*new_rows, *new_cols);
    if (!trans) return NULL;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            trans[j][i] = mat[i][j];
        }
    }
    return trans;
}

void print_matrix(double** mat, int rows, int cols) {
```

```
    for (int i = 0; i < rows; i++) {  
        for (int j = 0; j < cols; j++) {  
            printf("%.2f ", mat[i][j]);  
        }  
        printf("\n");  
    }  
    printf("\n");  
}
```

matrix_mult.h

```
#ifndef MATRIX_MULT_H  
#define MATRIX_MULT_H  
  
#include "matrix_utils.h"  
  
double** multiply_matrices(double** a, int a_rows, int a_cols, double** b, int  
b_rows, int b_cols, int* res_rows, int* res_cols);  
  
#endif /* MATRIX_MULT_H */
```

matrix_mult.c

```
#include "matrix_mult.h"  
  
double** multiply_matrices(double** a, int a_rows, int a_cols, double** b, int  
b_rows, int b_cols, int* res_rows, int* res_cols) {  
    if (a_cols != b_rows) {  
        printf("Error: Incompatible dimensions for multiplication.\n");  
        return NULL;  
    }  
    *res_rows = a_rows;  
    *res_cols = b_cols;  
    double** res = create_matrix(*res_rows, *res_cols);  
    if (!res) return NULL;  
    for (int i = 0; i < *res_rows; i++) {  
        for (int j = 0; j < *res_cols; j++) {  
            res[i][j] = 0;  
            for (int k = 0; k < a_cols; k++) {  
                res[i][j] += a[i][k] * b[k][j];  
            }  
        }  
    }  
    return res;  
}
```

Makefile for Compilation and Linking

This Makefile uses automatic dependency management (as explained previously) to handle headers properly. It compiles each .c to .o, generates .d files for dependencies, and links into an executable named matrix_app.
makefile

Compiler and flags

CC = gcc

CFLAGS = -Wall -Wextra -O2 -MMD -MP *# -MMD -MP for auto dependencies*

Executable name

TARGET = matrix_app

Source files (all .c except main if separate, but include all)

SRCS = main.c matrix_utils.c matrix_mult.c gauss_jordan.c linear_regression.c

Object and dependency files

OBJS = \$(SRCS:.c=.o)

DEPS = \$(SRCS:.c=.d)

Default target

all: \$(TARGET)

Link rule

\$(TARGET): \$(OBJS)

\$(CC) \$(CFLAGS) -o \$@ \$^ -lm *# -lm for math.h (fabs)*

Include dependency files

-include \$(DEPS)

Clean up

clean:

rm -f \$(OBJS) \$(DEPS) \$(TARGET)

.PHONY: all clean

How to Use the Makefile

1. Save all files in a directory.
2. Run make to build matrix_app.
3. Execute: ./matrix_app.
4. If you change a header (e.g., matrix_utils.h), make rebuilds all dependent .o files automatically.
5. Clean: make clean.

