MBARARA UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF COMPUTING AND INFORMATICS

COURSE UNIT: SOFTWARE ENGINEERING INDUSTRIAL MINI PROJECT II

COURSE CODE: SWE4106

Academic Year: 2024/2025

Semester: One

Student name:	**BWESIGYE TREASURE**
Regno:	**2021/BSE/145/PS**
Student number:	**2100603048**

Readme/Installation Guide

To use C.

Steps to use Notepad++ for C programming on Windows and a general approach for Windows.

Windows Setup with Notepad++ and C Compiler

1. Install Notepad++:

Download from [Notepad++] (https://notepad-plus-plus.org/downloads/).

Run the installer.(other text editors and ides(code blocks, intellij among others) exist but notepad gives a lightweight way to run code from different languages)

2. Install MinGW:

Download MinGW from [MinGW](https://osdn.net/projects/mingw/releases/).

During installation, select "mingw32-base".

Add the MinGW `bin` directory (e.g., `C:\MinGW\bin`) to your PATH.

3. Set Up Notepad++ for C:

Open Notepad++.

Go to *Settings* → *Preferences* → *Language* and select "C" for syntax highlighting.

4. Write a C Program:

Create and save a file named `world.c`:

Sample code

```
#include <stdio.h>


int main() {
    printf("Hello, World!\n");
    return 0;
}
```

5. Compile and Run:

Open Command Prompt.

Navigate to the code directory:

```
cd C:\path\to\your\file
```

Compile with:

gcc world.c -o world.exe

Run the program:

world.exe

Functions in my library include:

1. **mat_mult(double A, double B, double C, int n)** :
    - Performs matrix multiplication for two n x n matrices. The result is stored in the matrix C.
2. **dft(double complex input, double complex output, int n)** :
    - Computes the Discrete Fourier Transform (DFT) of the input array, producing the transformed output in the specified output array.
3. **gradient_descent(double (func)(double), double (grad)(double), double x, double lr, int steps)** :
    - Optimizes a given mathematical function using the gradient descent algorithm. It iteratively updates the value of x based on the learning rate (**lr**) and the gradient of the function for a specified number of steps.
4. **mat_mult_c(double A, double B, double C, int n)** (Wrapper function for C interoperability):
    - A wrapper function to call **mat_mult** for compatibility with C/C++ interfaces.
5. **dft_c(double complex input, double complex output, int n)** (Wrapper function for C interoperability):
    - A wrapper function to call **dft** for compatibility with C/C++ interfaces.
6. **gradient_descent_c(double (func)(double), double (grad)(double), double x, double lr, int steps)** (Wrapper function for C interoperability):
    - A wrapper function to enable calling **gradient_descent** easily from C/C++.

This list gives a clear overview of the key functions in your library, outlining their primary capabilities and usage.

Installation Steps

Building the Library

Clone the repository:

https://github.com/treasure16522/Portable-library.git

cd Portable-library

Compile the library:

For Linux/macOS:

gcc -shared -o libmatrix.so -fPIC mylibrary.c

For Windows:

gcc -shared -o matrix.dll mylibrary.c

Using in Different Languages

Python

Install ctypes if not already available.

Use this example:

```
import ctypes

lib = ctypes. CDLL('. /libmatrix.so')
lib.mat_mult.restype = None
# Example usage
```

Rust

Add to Cargo.toml:

```
[dependencies]
libc = "0.2"
```

Use libc to call functions.

C++

Include the header file:

```
extern "C" {
void mat_mult(double A, double B, double C, int n);
}
```

Link with the shared library during compilation.

Steps to Use the Library in Java

1. Create the Native Library

Compile your C library into a shared object or dynamic link library:

On Linux:

bash

Copy code

```
gcc -shared -o libmatrix.so -fPIC mylibrary.c
```

On Windows:

cmd

Copy code

gcc -shared -o matrix.dll mylibrary.c

2. Write a Java Wrapper Class

In Java, you create a wrapper class that uses System.loadLibrary() to load your shared library at runtime.

You declare the native methods in the Java class using the native keyword.

3. Generate JNI Headers

Use the javac compiler to compile your Java wrapper class and generate a .class file.

Use the javah tool (or its equivalent in modern JDKs, like javac -h) to generate a JNI header file. This file defines the interface for Java to call your native methods.

Example Command:

bash

Copy code

javac -h . WrapperClass.java

4. Implement the JNI Functions

Implement the JNI functions in C. These functions will bridge the calls between Java and your existing library functions.

For example:

mat_mult in Java will map to Java_PackageName_WrapperClass_matMult in C.

5. Compile and Link the JNI Implementation

Compile the JNI implementation along with your library to ensure seamless integration.

6. Run the Java Program

Set the java.library.path system property to include the directory where your shared library is located:

On Linux:

bash

Copy code

java -Djava.library.path=. YourJavaProgram

On Windows:

cmd

Copy code

```
java -Djava.library.path=. YourJavaProgram
```

To run the provided test cases in C programming language (fourier.c and matrix.c) using my library (mylibrary.dll on Windows or mylibrary.so on Linux/macOS), you can follow these steps:

1. Share the Compiled Library Only

Provide the shared object file (mylibrary.so for Linux/macOS or mylibrary.dll for Windows) without sharing the source code (mylibrary.c).

Share the mylibrary.h header file, as it defines the function prototypes required for test cases to use the library.

2. Setup Environment

Linux:

Place the mylibrary.so file in a known directory, e.g., /usr/local/lib or the current directory.

Ensure the directory containing mylibrary.so is in the library path:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Windows:

Place the mylibrary.dll file in the same directory as the test executable or in a directory listed in the system's PATH variable.

3. Compile Test Cases

Use the gcc compiler to compile the test cases (fourier.c and matrix.c) into executables. Link against the shared library without needing the library's source code.

Linux:

```
gcc -o fourier_test fourier.c -L. -lmylibrary -lm
```

```
gcc -o matrix_test matrix.c -L. -lmylibrary -lm
```

Windows:

```
gcc -o fourier_test.exe fourier.c mylibrary.dll
```

```
gcc -o matrix_test.exe matrix.c mylibrary.dll
```

4. Run the Test Cases

After compiling, run the test cases, ensuring the shared library is accessible.

Linux:

```
./fourier_test
```

./matrix_test

Windows:

fourier_test.exe

matrix_test.exe

5. Verify Results

The outputs of the test cases (e.g., matrices for matrix.c, transformed data for fourier.c) will verify the functionality of *mylibrary.dll or mylibrary.so*.

Since the test executables link to the compiled library, the library's internal source code (mylibrary.c) remains hidden from the user.

Key Points:

Binary Distribution: By providing only the compiled .so or .dll files, you retain ownership of your source code.

Interface Sharing: Only share the mylibrary.h file to allow test cases to interact with the library.

Cross-Platform Use: Provide both .so and .dll versions for compatibility across Linux and Windows.