Homework #7 - Linear Equations Solver

Objective: Develop a C module for matrix computations and solution of Linear Equations using Gaussian Elimination. Optimize it and time it against a similar solver implemented in the GSL library.

The Problem

The file **hw7_files.tar** has a partial implementation of the module (**apmatrix.h, apmatrix.c**) for matrix computations that you will extend to solve systems of linear equations using Gaussian elimination with partial pivoting, more precisely, *permuted LU factorizations*. It also has a file, **gsl_example.c**, that shows how to use GSL to solve this problem. The objective is to develop an implementation of the Gaussian Elimination algorithm with pivoting, as demonstrated in class, and optimize it to how close you can come to the GSL algorithm.

You must use the prototypes provided in the header file **apmatrix.h**, they have been chosen to make your interface similar to the one provided in GSL (examine the code in **gsl_example.c**).

Module Implementation

Following is a list of prototypes for the functions that you need to implement. For more details see **apmatrix.h**. Feel free to add more "helper functions" if this simplifies the development and debugging of your module and document your additions/changes in **analysis.txt**.

1. **rVector* rv_alloc(int n);**
2. **iVector* iv_alloc(int n);**
3. **void rv_free(rVector* V);**
4. **void iv_free(iVector* V);**
5. **void PLU_factor(Matrix* A, iVector* p);**
6. **void PLU_solve(Matrix*A , iVector*p, rVector* b, rVector* x);**

The last two functions implement the linear equations solver; the elimination step using in-place Gaussian Elimination with partial pivoting will be implemented in **PLfactor(Matrix* A, iVector* p)** and the solution (forward and backward substitution) step using the modified matrix *A* and the permutation vector *p* of the elimination step will be implemented in **PLU_solve(Matrix*A , iVector*p, Vector* b, Vector* x)**.

These functions were designed to parallel the GSL solver (examine the **gsl_example.c** code)

Testing

To test your module you will write a driver program called **hw7.c** and also write a **makefile** to facilitate testing and debugging. Your hw7.c driver will follow the same syntax as hw6.c.

Your driver should satisfy the following requirements:

e.g. ./hw7 -input **matrix.txt** - Will solve matrix.txt without any detailed debug information
./hw7 -in **matrix.txt** **-verb** - Will solve matrix.txt producing detailed debug information including: P and LU values.

LU may be separate or combined. E.g.:

```
*************** GE0.txt **************
Solves for Ax=b using PLU factorization.
A =
 -20.0000   55.0000  -10.0000
 -10.0000  -10.0000   50.0000
  30.0000  -20.0000  -10.0000

b =
          0.0000
        200.0000
          0.0000

P = [ 2 0 1 ]
LU matrix =
  30.0000  -20.0000  -10.0000
  -0.6667   41.6667  -16.6667
  -0.3333   -0.4000   40.0000

x =
          3.0000
          2.0000
          5.0000
```

a) It should be able to work for any system that has **as many equations as unknowns**, e.g., the *A* matrix should be square. Otherwise it should stop and print a suitable message to **stdout**.

b) The input should be read from a text file formatted as follows: The first line will have two numbers (separated by white space), specifying the dimensions of the augmented matrix $G = [A \; \boldsymbol{b}]$, the following lines will have the rows of *G*, each entry separated by spaces. The example below shows the two first lines of the data file for a system of 5 equations with 5 unknowns

5 6
1.1 3.4 5.6 7.8 7.8 1.1
...

Note that the first 5 entries in the 2nd line correspond to the first row of *A* and the last entry to the corresponding entry in ***b***.

Your input data will not be "nice" (aka. Real world data). The data will have tab, carriage control, line feed, trailing space or end-of-file characters in it. Your parser should "do the right thing", e.g handle all the corner cases.

c) If the system of equations **do not have a solution** the program should exit gracefully and report it, printing a suitable message to **stdout**.

d) If the system has a solution the program should print the solution to stdout. If a verbose option was specified at the command line then it should also print the matrices ***L,U*** and the permutation vector ***p***, regardless of the existence of a solution. ***L,U*** may be individual matrices or a single combined matrix.

e) Your program should write a suitably *formatted* output to **stdout** (so that it can be redirected, if desired, to a text file).

Test your module with the test cases (geX.txt) included in **hw7_files.tar**. Analyze the results and include them in the file **analysis.txt**. Also generate a file called **out.txt** to include the output for the test cases given (I will test your programs with other data sets to make sure they matrices of arbitrary size).

Optimization

Once your program is working correctly you will compare its performance by timing it (using **Timers.h**) against the GSL solver. To get meaningful timing just use large matrices as test inputs (recall that elimination is $O(n^3)$ and solution is $O(n^2)$ so you can easily find an n where the times are non-negligible). You do not need to loop your code, just use a large data set.

a) Begin by taking the sample program **gsl_example.c** and modifying it to take inputs from the test files (formatted as explained above) and write output to **stdout**. Name the modified program **gsl_test.c**. Also include timing in this program using *two timers*, *one for the elimination and one for the solution step.* First test **gsl_test.c** with the same (small) test matrices that you used to test the module to show your implementation give the correct answers.

Compile command: gcc -ansi **-lgsl -lgslcblas** gsltest.c -o glstest

b) You have been provided with a large matrix (**2047 2048)** of data called **rand.txt** to use as data for timing your implementation against GSL's (the important point here is to choose matrices that are large enough).

c) Add timing instructions to **hw7.c** and **gsl_test.c** , then record your timing for both. Compute as a performance metric the ratio the time that your implementation took over the time that GSL took. Your code must use the compile line option **–DEN_TIME** to enable and disable the timing function. You do NOT need to fine tune your implementation.

Makefile:

You must provide a quality Makefile with the following targets: all, test, gsl, mem, help and clean.

"all"	-should make **hw7** and **gsl_test.**
"test"	- should run **hw7** with **each geX.txt** file**,** in turn, all redirected to a single file: **out.txt.**
"gsl"	- should run **gsl_test** with **each geX.txt** file**,** redirected to **out.txt.**
"mem"	**-** should run **hw7** with **ge2.txt** using **valgrind** redirected to **mem.txt**
"time"	- should run **gsl** and **hw7** with timing enable using **rand.txt**, redirected to: **out.txt.**
help, clean	- should do the normal things

Note: Some geX.txt files do not have a solution and will generate an error which will STOP make. Use the "-" make option to force the commands to continue execution

Submission

Pack your files in the tar file **lastName_hw7.tar** (lastName is your last name) and submit it. Include in the tar file your C programs, the makefile, your analysis and any additional data that you used for testing your programs. Grading Criteria

1. (15 points) Miscellaneous
   (a) (15 points) makefile works properly

2. (65 points) Accuracy and Robustness
   (a) (10 points) No memory leaks,
   (b) (15 points) Program handles arbitrary matrices, stops and prints message if no solution.
   (c) (20 points) Algorithm give correct solutions.
   (d) (20 points) Correct timing approach for module and GSL test.

3. (20 points) Analysis and Results
   (a) (5 points) Computation results clearly formatted.
   (b) (5 points) Appropriate timing results included.
   (c) (10 points) Concise and clear analysis.

Note: Not all the test data is necessarily correct or solvable, remember data can be bad, that's why we code defensively