**1.** Compose a program to multiply two NxN matrices in C.

See "matrixTemplate.c" and "matrixTemplate.h".

**2.** Create a script to generate six programs that will multiple the same matrices with identical data in six different (i,j,k) orders.

See "program_generator.py". I wrote a Python script to generate the six function of different (i,j,k) orders.

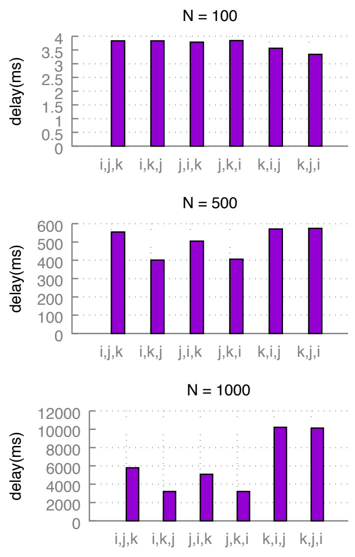
**3.** Verify that the outputs of six different matrix programs are identical.

There is an int type varibale `check` in the "main.c". The program will compare the result of the six functions if `check` is set to 1.

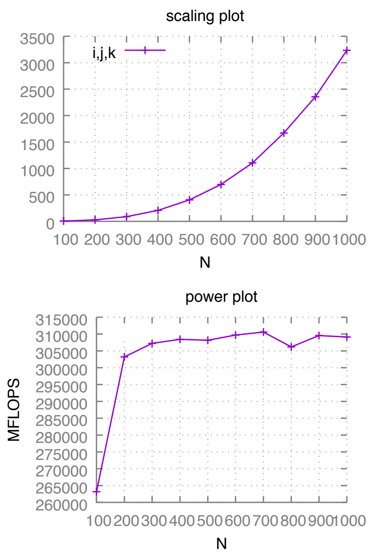
**4.** Compose a script to harvest the running times of N=500 for the six different programs. Find the best performing order. Then run the best performing order for N=100, 200, 300, .., 1000.

I use a loop to let the program run for all the N values.

**5.** Plot your results with explanations. There are two investigations: a) performance behavior for six orders, and b) performance behavior for different N.



(a) different i-j-k order



(b) scaling

Figure 1

Figure. 1 a shows the delay to do the matrix multiplication for the six i-j-k orders and Figure. 1 b shows the delay for the different N values.

We can see that the order (i,k,j) and (j,k,i) have the best performance while the order (k,i,j) and (k,j,i) are worst. The difference is due to the special locality. When the matrix multiplication is calculated by the order (i,k,j) or (j,k,i), the program calculates the elements in the results one by one, without running back and forth, so the memory block holding the row or the column would not be loaded into CPU cache repeatedly. As for the order (k,i,j) and (k,j,i), they calculate all the elements of the result matrix at the same time, i.e., they loop over the memory blocks of the result matrix again and again until the calculation for all the elements has been calculated. And thus the memory block holding the row or column are loaded in to cache repeatedly. To sum up, each element in the result matrix requires **N multiplication operations**, and the order (i,k,j) and (j,k,i) will run the **N multiplication in a row**, while the the order (k,i,j) and (k,j,i) run **one multiplication each time for each element**, incurring overhead in reloading memory blocks holding the row and columns of the two multiplying matrices.

As for the scaling, the order (i,k,j) is chose to run the program. As the N grows, the running times grows by the squere of N. And we can see that the computing power remains quite stable except for the first point where N = 100, which probably because the matrix is too small and thus the program's process is ended before it is elevated to higher priority.