Why MATLAB?

Once upon a time computers were slow and unreliable and all programming was done in machine or assembly code that corresponded directly to the operations that could be performed by the central processing unit (CPU) and the floating-point processing unit (FPU). Assembly code for a program that calculated $z \leftarrow xy$ might look, in a poetic sort of way, like

Code Example 1a

```
fload r1,@x
fload r2,@y
fmult r1,r2
fsave @z,r1
```

There are two problems with the assembly code: First, it is difficult to check whether such a code performs a desired calculation. Second, assembly code requires the programmer to specify details irrelevant to the calculation to be performed. In particular, the above example specifies that the registers `r1` and `r2` should be used to store the variables x and y, respectively, during the calculation. Over specification of irrelevant details for a calculation can prevent certain mathematical optimizations from being employed automatically by an optimizing compiler.

Although the importance of good algebraic notation in mathematics had been recognized for hundreds of years prior to the manufacture of the first digital computer, it took more than 10 years before the first high-level programming languages became practical.

FORTRAN, short for FORmula TRANslator, was developed in 1954 by John Backus at IBM and originally called the IBM 701 Speedcoding System. Though it is not quite the first high-level programming language, it is surely the most successful. Fortran 77 is widely used to code numerical algorithms for engineering and scientific applications. The Fortran code for $z \leftarrow xy$ is

Code Example 1b

```
      z=x*y
```

Some of the success of Fortran comes from tradition. Large libraries of existing well debugged Fortran code make it convenient to write new programs in Fortran. Moreover, the specific semantics of Fortran often allow an optimizing compiler to better analyze data flow within a Fortran program than could be done for the corresponding program written in C. This results in a more efficient register allocation, vectorization and corresponding faster run times of Fortran code. New features have been added to Fortran only with careful consideration on how they will affect the automatic optimization of numerical codes. Fortran 90 supports vector operations, operator overloading and modules which contain both data and code. Modern Fortran 2000 supports object oriented programming in a way similar to Java and C++.

Two of the best known numerical subroutine libraries written in Fortran are EISPACK and LINPACK. The subroutines in EISPACK solve numerical eigenvalue problems and the subroutines in LINPACK are for solving systems of linear equations. In 1979 Cleve Moler, one of the original authors of LINPACK, created MATLAB, a programming language whose builtin algebraic operations included those found in EISPACK and LINPACK.

Consider the Fortran code for performing the $n \times n$ matrix multiplication given by $C \leftarrow AB$. A naive implementation of this operation might look like

Code Example 2a

```
      do 30 i=1,n
          do 20 j=1,n
              sum=0.0
              do 10 k=1,n
                  sum=sum+A(i,k)*B(k,j)
10            continue
              C[i,j]=sum
20        continue
30    continue
```

It is difficult to tell that this Fortran code performs a matrix multiplication. Moreover, the Fortran code over specifies details of the matrix calculation. In particular, it specifies a particular order in which the constituent multiplications are to be performed. These problems are identical to the problems we discussed for Example 1a, the assembly language version of the scalar multiplication $z \leftarrow xy$. The Matlab code for $C \leftarrow AB$ is

Code Example 2b

```
C=A*B
```

By specifying only the necessary details, Matlab is free to use any algorithm optimal for the particular hardware when performing the matrix multiplication.

Let us compare the execution times of the naive Fortran code to the Matlab code for a 1024×1024 matrix. The hardware for this test is a 1.7Ghz Intel P4 with a 256K cache size and PC133 main memory. The compiler used was gnu Fortran version 3.4.4 with optimization flags `-O3`.

$n \times n$	Example 2a	Example 2b
1024×1024	174.84s	2.17s

MATLAB can also optimize the way in memory the matrices are stored. For example, MATLAB has special ways of storing matrices that have a large percentage of zero entries. Let us discuss two obvious ways to store dense matrices whose entries are all non-zero in computer memory. For this discussion $\texttt{RAM}[k]$ will denote the k-th memory location in main memory capable of storing one scalar entry of a matrix. Consider an $n \times n$ matrix A with entries A_{ij}. The first way to store A is called column-major format and was used by IBM in their Fortran compilers.

$$A_{ij} = \texttt{RAM}[(i-1) + n(j-1)]$$

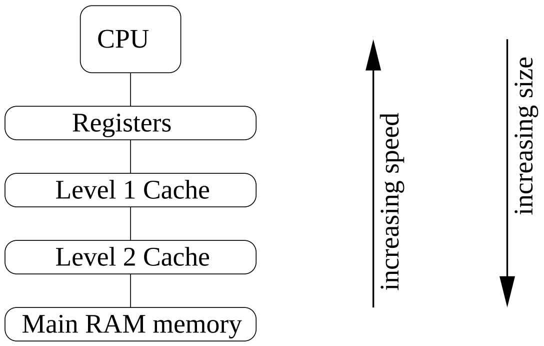
The second way to store A is called row-major format and was used by DEC in their Fortran compilers.

$$A_{ij} = \texttt{RAM}[n(i-1) + (j-1)]$$

DEC was purchased by Compaq and subsequently sold to HP. DEC's processor design team now works for Intel. Fortran 77 and greater has standardized on column-major format for storing two-dimensional arrays. C/C++ uses row-major format.

Why does the Matlab version of $C \leftarrow AB$ run so much faster than the Fortran version? To understand what possible ways Example 2a could be optimized, it is necessary to understand a little about how a computer accesses memory. A physical reality is that, as a memory device gets larger, it tends to get slower. Cache memories are very fast but are also small and expensive. Main memory is inexpensive and large, but is slow.

Memory Hierarchy



A primary factor determining of the speed at which $C \leftarrow AB$ can be computed is how fast the data can be fetched from memory. To optimize this, the algorithm can be rewritten to keep as much of the data in fast cache memory as possible during the calculation. In effect one breaks the matrix into smaller submatrix blocks, each of which fit into cache memory. These matrix blocks are then multiplied separately in cache. This optimization technique is called cache blocking. More information about cache blocking as well as code examples may be found by Googling on `cache blocked matrix`.

Another thing to keep in mind is that data prefetch mechanisms and various cache replacement policies paradoxically make it more efficient to access RAM or random access memory in sequential order. Assuming we are using a Fortran compiler that stores matrices in column-major order, we can interchange the order of the loops in Example 2a to obtain

Code Example 2c

```
      do 2 j=1,n
          do 1 i=1,n
              C(i,j)=0
1         continue
2     continue
      do 30 k=1,n
          do 20 j=1,n
              do 10 i=1,n
                  C(i,j)=C(i,j)+A(i,k)*B(k,j)
10            continue
20        continue
30    continue
```

This simple change results in a significant increase in speed of execution, but is still noticeably slower than the Matlab version.

If we goof and optimize as if the compiler were using row-major format for storing the matrices, then we obtain

Code Example 2d

```
      do 2 i=1,n
          do 1 j=1,n
              C(i,j)=0.0D0
1         continue
2     continue
      do 30 k=1,n
          do 20 i=1,n
              do 10 j=1,n
                  C(i,j)=C(i,j)+A(i,k)*B(k,j)
10            continue
20        continue
30    continue
```

Example 2d runs really slow when compiled with a standard Fortran compiler that stores matrices in column-major format. The results are summarized as

$n \times n$	2a	2b	2c	2d	2e
1024×1024	174.84s	2.17s	19.88s	441.60s	1.08s

To achieve the speed of Matlab we would need to use a cache blocked matrix multiply and carefully employ the data prefetch and sse2 short vector instructions of the Pentium 4 processor. The optimal matrix blocking size depends on the size and relative speed of the cache memory compared to main memory. These hardware specific parameters vary between computer architectures and processor model.

We end by noting that the matrix multiplication operations in modern versions of Matlab are performed by ATLAS, which stands for Automatically Tuned Linear Algebra Software. ATLAS is free software written by Clint Whaley, Antoine Petitet and Jack Dongarra at the Innovative Computer Laboratory at the University of Tennessee. Google on `ATLAS Linear Algebra` for more information.

The core of ATLAS is a specialized compiler for vector and matrix multiplications which outputs code written in a combination of C and assembly language. This compiler is coupled with an empirical optimizer which adjusts the matrix blocking size and other parameters fed into the compiler to search for the fastest version of the algorithm for a particular hardware. ATLAS supports what is called BLAS levels 1, 2 and 3. There are vector-vector, vector-matrix and matrix-matrix operations, respectively. The operation $C \leftarrow AB$ written directly as a BLAS level 3 subroutine call looks like

Code Example 2e

```
      call dgemm('N','N',n,n,n,1.0D0,a,n,b,n,1.0D0,c,n)
```

This code runs even faster than the Matlab code; however, it is syntactically less obvious what it does. Such a call could be useful when most of the code has already been or must be written in Fortran or C. The difference in execution time between 1.08s and 441.60s should not be missed.