혼합 컴파일과 스크립트 언어

The Mixcd/Language Compilation and Script Languages

조 정 래*

*한국건설기술연구원 구조연구부 선임연구원

1. 들어가면서

이번 연재에서 다룰 내용은 하나의 프로그램을 작성할 때 여러 컴파일러나 프로그래밍 언어를 혼합하여 완성하는 것(혼합컴파일, Mixed Compiler/Language Compilation)에 대해 다룹니다. 주 목적은 지난번 연재에서 소개한 GCC 환경에서 컴파일된 ATLAS, LAPACK, CHOLMOD를 대상으로 Microsoft Visual C/C++나 Fortran 등 여러 컴파일러 및 언어에서 호출하는 방법을 소개하는 것입니다. 마지막으로 인터프리터 언어(interpreter language) 등으로 불리기도 하는 스크립트 언어(script language)를 소개합니다.

2. 혼합 컴파일(Mixed Compiler/ Language Compilation)

여러 언어를 섞어서 프로그램을 완성하는 것은 구조해석프로그램을 작성할 때 사용되는 일반적인 방법입니다. 그런데 이 연재에서 왜 Mixed Compilier라는 용어를 추가 했을까요? 그 이유는 Win32 환경에서 주로 사용되는 대표적인 두 가지 컴파일러 계열이 있고, 두 계열 사이에서 오브젝트 파일(object 파일)이나 라이브러리, DLL 등이 경우에 따라서 호환되지 않을 수 있기 때문입니다. 두 계열 중 하나는 Microsoft 사에서 개발한 C/C++용 Microsoft Visual C/C++(이하 MSVC)와 포트란용 Digital Visual Fortran(이하 DVF) 등이고, 다른 하나는 GNU GCC를 Win32용으로 사용할 수 있게 한 MinGW, CygWin 등이 있습니다. GCC는 GNU Compiler Collectin의 줄임말로 C 소스 외에 C++, 포트란 등과 같은 다양한 언어를 컴파일 하는데 사용할 수 있습니다(C/C++ 컴파일할 때는 gcc, 포트란을 컴파일할 때는 g77 사용). 일반적으로 수치라이브러리를 개발하는 사람들은 Unix/Linux에 익숙한 사람들이 많고, 특히 free software를 지향하는 사람들의 경우 GCC를 애용합니다. 따라서 GCC 등에서 개발한 라이브러리를 MSVC나 DVF에서 사용하기 위해서는 여러 작업이 필요할 수 있습니다. 여기에서 필요한 작업은 컴파일러가 다르기 때문에 고려해주어야 할 사항과 C/Fortran 등 언어가 다르기 때문에 고려해주어야 할 사항 등이 혼재되어 있습니다. 이런 이유로 이 연재에서는 Mixed Compiler라는 용어를 추가하였습니다.

이번 연재에서는 CygWin의 GCC를 사용하지 않고 MinGW라는 Win32용 GNU GCC를 사용하겠습니다. 지난번 연재에서 사용한 CygWin의 GCC는 CygWin이라는 Linux 에뮬레이션 환경에서 구동되던 것과 달리 MinGW는 Windows 환경에서 구동되기 때문에 보다 편리하게 사용할 수 있습니다. 공식 홈페이지는 www.mingw.org이고, www.sf.net에서 MinGW를 검색하면 MinG-5.1.2.exe와 같은 인스톨러를 다운받을 수 있습니다. 물론 CygWin에서 -no-cygwin 옵션을 사용하면 MinGW를 사용한 것과 동일하기 때문에 CygWin을 사용해도 무방합니다.

이제 혼합 컴파일에서 필요한 개념 먼저 정리하고, 전번 연재에서 컴파일한 ATLAS, LAPACK, CHOLMOD를 여러 환경에서 사용하는 방법을 알아보기로 합니다.

2.1 기본 개념

혼합 컴파일을 하기 위해서 지켜야할 약속이 5가지가 있습니다. 오브젝트 파일에서의 symbol(함수나 상수)의 이름이 어떻게 바뀌나를 의미하는 이름규약(naming convention), 스택과 관련된 호출규약(calling convention), 함수의 인자가 전달되는 방식(parameter passing rule)과 그 데이터 형, runtime library 등입니다. 이 규약들은 언어마다 다른 뿐아니라 컴파일러마다 다를 수 있습니다. 즉, C 소스를 컴파일할 때 GCC를 쓰느냐 MSVC를 쓰나냐에 따라서도 달라지게 됩니다. 이 때문에 동일한 언어를 작성된 소스라 하더라도 컴파일러가 다르면 링크하지 못할 수 있습니다.

(1) 이름규약(naming convention)과 호출규약(calling convention)

이름규약(naming convention)은 소스를 컴파일해서 오브젝트 파일(.obj화일)을 만들면, 심볼(함수이나 상수 등)의 이름이 변형되어 저장되는데, 이때 사용되는 규약을 의미합니다. 예를 들어 포트란으로 작성된 subroutine Add(A,B,C)는 G77로 컴파일할 때 _add로 바뀌고, DVF로 컴파일할 때 _ADD로 바뀝니다. 따라서, G77로 컴파일된 Add함수를 DVF에서 호출하면 링크시에 함수자체를 인식하지 않게 됩니다. 호출규약(calling convention)은, 함수를 호출할 때 스택을 이용해 함수의 인자를 넘겨주게 되는데, 스택으로 넘겨주는 인자들의 순서와 호출이 끝난 후 스택을 청소하는 주체를 정의하는 규약입니다. 대표적인 규약으로는 __cdel과 __stdcall 등이 있습니다. __cdel은 인자를 오른쪽에서 왼쪽으로 넘겨주고, 호출이 끝나후 스택의 청소는 호출하는 함수가 책임집니다. __stdcall은 스택 청소를 호출되는 함수가 책임진다는 점에서 다릅니다. __cdel은 C언어의 표준으로 GCC, MSVC에서 공통으로 사용되며, __stdcall은 Win32 API 함수에서 적용되는 규약입니다. 포트란의 경우 표준 호출규약이 없는데, DVF에서는 __stdcall을 사용하고, G77에서는 __cdel을 사용합니다. 호출규약이 일치하지 않을 경우 링크는 되지만 프로그램 실행시 스택에 오류가 발생하여 프로그램이 다운되는 것과 같은 심각한 오류를 발생할 수 있습니다. 표 1은 컴파일러 및 언어에 따라 사용되는 이름 및 호출 규약을 요약한 것입니다. 표에서 이름 및 호출 규약이 컴파일러나 언어에 따라 밀접하게 연관이 되어 있음을 알 수 있습니다. C 언어의 경우 디폴트 호출규약은 __cdel이지만, __stdcall 키워드를 이용해 호출규약을 소스에서 바꿀 수 있습니다. 즉,

```
int __stdcall Sum(int a,int b,int c);
```

와 같이 작성하면 __cdel이 아닌 __stdcall 규약을 사용하게 됩니다. 표에서 @n이 적용되는 경우의 의미는 _Sun@12와 같이 사용되는 인자의 크기(여기서는 4바이트 크기의 인자가 3개 사용되었음) 만큼 @ 뒤에 숫자를 기입하는 형식을 의미합니다.

표 1 포트란 및 C의 호출 규약에 따른 이름규약(함수명이 Function일 경우)

언어 또는 컴파일러	Calling convention	.Obj 파일 내 이름	비고
C _cdel	__cdel	_Function	MSVC, GCC(MinGW)
C __stdcall	__stdcall	_Function@n	
DVF	__stdcall	_FUNCTION	
G77	__cdel	_function_	

이름 및 호출 규약은 혼합컴파일에서 가장 중요한 문제입니다. 따라서 간단한 예를 통해 확인해 보기로 합니다. 리스트 1과 같은 예제 파일(cadd.c와 fadd.f)를 작성한 후 컴파일해서 dumpbin이라는 유틸리티로 확인해 보겠습니다.

리스트 1. 이름 규약의 확인 예

C(cadd.c)

```
void Add1(double* a,double* b,double* c) {*c=*a+*b; }
void __stdcall Add2(double* a,double* b,double* c) {*c=*a+*b; }
```

MSVC에서 확인

```
> cl /c cadd.c
> dumpbin /symbols cadd.obj
   .... _Add1, _Add2@12 이 출력됨을 확인할 수 있음
```

GCC에서 확인

```
> gcc -c cadd.c
> dumpbin /symbols cadd.o
   .... _Add1, _Add2@12 이 출력됨을 확인할 수 있음
```

Fortran(fadd.f)

```
      SUBROUTINE Add(A,B,C)
      C = A+B
      END
```

DVF에서 확인

```
> df /c fadd.f
> dumpbin /symbols fadd.obj
  ... _ADD이 출력됨을 확인할 수 있음.
```

G77에서 확인

```
> g77 -c fadd.f
> dumpbin /symbols fadd.o
  ... _add_ 이 출력됨을 확인할 수 있음.
```

(2) 함수 인자 관련 사항

이종의 언어로 작성된 함수를 연결할 때는 함수의 인자(argument)와 관련해서 고려해야 할 점이 두가지 있습니다. 하나는 함수에서 인자를 전달하는 방식(argument passing rule)을 고려해 주어야 하고, 다른 하나는 데이터 형을 일치시켜야 한다는 점입니다. 포트란은 인자의 포인터를 넘기는데(call by reference) 반해, C는 인자의 값을 복사해서(call by value) 넘기게 됩니다. 따라서, C와 포트란을 연결하기 위해서 일반적으로 C의 포인터를 이용하게 됩니다. 데이터형의 일치라는 것은 포트란에서는 부동소수점을 다루는 4바이트 변수가 real로 정의되지만 C에서는 float로 정의하는데, 이와 같이 서로 대응되는 데이터형을 사용하는 것을 의미합니다. 표 2는 C측에서 포인터를 사용한다고 가정했을 때 주요 내장 변수형의 대응 관계를 나타낸 것입니다. 예를 들어 8바이트 크기로 할당되는 포트란 Real*8은 C의 double 형 포인터 double*로 대응시켜야 합니다.

표 2 포트란과 C/C++간 함수의 인자

Fortran	C/C++	비고
CHARACTER*1	unsigned char*	1byte (unsigned)
INTEGER 또는 INTEGER*4	int*	4byte (signed)
REAL 또는 REAL*4	float*	4byte (signed)
DOUBLE PRECISION 또는 REAL*8	double*	8byte (signed)

마지막으로 C와 포트란을 혼합할 때에는 포트란의 Function과 같이 값이 리턴되는 경우는 쓰지 않는 것이 좋습니다. 왜냐하면 대부분의 수치라이브러리는 값이 리턴되는 Function을 쓸 경우가 별로 없고, 또한 이 경우 고려해야할 문제가 또 생기기 때문입니다. 또한, 포트란과 C의 문자열을 처리하는 방식이 다르기 때문에 이것 역시 사용하지 않는 것이 좋습니다. 이외에도 포트란에서만 지원하는 complex형, logical형, common문 등은 사용하지 않는 것이 좋지만, 반드시 필요하다면 컴파일러에 설명된 문서를 참조하시기 바랍니다.

이제 예를 들어보기로 하겠습니다. G77과 GCC를 사용하는 환경에서 포트란에서 다음과 같은 함수가 있다고 가정하면,

```
SUBROUTINE add(A,B,C)
REAL*8 A,B,C
C = A+B
END
```

이 함수는 C에서

```
void add_(double* A, double* B, double* C)
```

와 같은 함수를 정의하고 사용하는 것과 같습니다. 위 예는 포트란을 컴파일하는데 G77를 사용하기 때문에 C에서 정의하는 함수가 _가 추가로 붙게 됩니다.

(3) 런타임 라이브러리(Runtime library)

런타임 라이브러리는 각 언어에서 정의되는 저수준의 루틴을 모아놓은 라이브러리를 의미합니다. 예를 들어 printf와 같은 C 언어의 입출력 함수는 MSVC에서 정적 라이브러리(static library)로 링크할 경우 libc.lib라는 라이브러리 파일에 정의되어 있고, 동적라이브러리(dynamic linked library)로 링크할 때는 msvcrt.dll에 정의되어 있습니다. 일반적으로 오브젝트 파일은 자신이 링크될 라이브러리에 대한 정보를 가지고 있지 않기 때문에 큰 문제가 되지 않습니다(MSVC로 컴파일된 오브젝트를 이 정보를 포함하고 있음. 따라서 여러 컴파일러에서 사용하기 위해서는 오브젝트를 만드는 컴파일 단계에서 이를 제거해야 함). 문제가 되는 점은 포트란으로 오브젝트를 만들어 C/C++에서 호출하는 경우입니다. 포트란은 입출력문을 라이브러리 함수로 처리하는 것이 아니라, WRITE나 READ, OPEN 등과 같은 포트란 언어의 키워드로 처리하게 됩니다. 이 키워드들은 결국 오브젝트 파일 내에서 어떤 컴파일러에 의존적인 함수로 변환되게 됩니다. 따라서 이 문장을 가지는 포트란 오브젝트 파일을 링크할 때는 그 컴파일러에 의존적인 라이브러리를 같이 링크해 주어야 하는 문제가 발생합니다. 예를 들어 G77의 경우 내부적으로 포트란 소스를 C 소스로 변환해서 오브젝트 파일을 생성합니다. 따라서 포트란으로 작성된 오브젝트를 링크할 때는 libg2c.a라는 라이브러리 파일이 필요한데, 이 라이브러리가 앞서의 입·출력문과 연관된 내부 모듈을 갖게 됩니다. 결과적으로 얘기하면, 포트란으로 라이브러리를 만들어 C에서 호출해서 사용할 때는 Read, Write문과 같은 입출력문을 사용하지 않거나 그 사용을 최소화해야 한다는 점입니다. 예를 들어 일반적으로 포트란을 이용한 라이브러리 개발자는 이러한 문장을 라이브러리내에서 하나의 함수로 모읍니다. 예를 들어 LAPACK이나 BLAS의 경우 xerbla.f라는 에러 출력 루틴에서만 WRITE문을 쓰고 있습니다. 결과적으로 이 루틴만 C 루틴으로 대체해 주면 C에서 이상없이 호출하여 사용할 수 있습니다.

(4) 예제

이제 예제를 통해 앞서에서 언급한 내용을 실습하기로 합니다. 여기에서 소개한 내용은 G77이나 GCC로 작성된 오브젝트를 MSVC, DVF에서 사용하는 방법을 중점적으로 설명하기로 합니다.

C언어로 작성된 함수를 GCC로 컴파일해 오브젝트로 만든 후 MSVC로 호출해서 사용하거나 그 반대의 경우는 같은 C언어로 작성하는 것이기 때문에 문제없이 작동합니다. 하지만, 포트란은 G77과 DVF간 이름 및 호출규약이 다르기 때문에 고려해야 할 점이 생기게 됩니다. 예제로 주어진 첫 번째는 G77로 작성한 예제를 DVF에서 호출하여 사용하는 예입니다. 리스트 2는 예제로 사용할 두 함수를 정의하고 있습니다.

리스트 2. G77과 DVF간 혼합컴파일

```
C add.f
      SUBROUTINE ADD(A,B,C)
      REAL*8 A,B,C

      C = A+B
      END

C caller.f
      PROGRAM TEST
      REAL*8 A,B,C
      A = 3.
      B = 1.
      CALL ADD(A,B,C)
      WRITE(*,*) 'A,B,C =',A,B,C
      END
```

G77과 DVF는 표 1에 나타낸 것과 같이 호출규약과 이름규약이 다르게 됩니다. 또한 포트란에서 호출규약이나 이름규약을 바꿀 방법이 없습니다(정확하게 얘기하면 DVF에서는 가능). 따라서 여기에서는 C언어로 중간에 wrapper 코드를 작성하는 방법으로 호출이 가능하도록 하였습니다. 리스트 3는 G77로 작성된 루틴을 DVF에서 호출하는 예제입니다. 중간에 wrapG77toDVF.c 라는 같은 wrapper 코드를 사용하고 있습니다. 이런 wrapper code는 C와 포트란 간의 호출에서도 많이 사용되는 기법입니다.

두 번째 예제는 G77로 컴파일된 포트란 함수를 GCC나 MSVC에서 불러쓰는 방법입니다. 리스트 4는 G77로 작성된 add 함수를 GCC나 MSVC에서 호출하여 사용하는 예입니다. 호출하는 쪽에서 함수이름만을 주의해서 호출하면 됩니다.

리스트 3. DVF에서 G77 오브젝트를 호출하는 방법

```
/*wrapG77toDVF.c */
extern void add_(double* a,double* b,double*c);

void __stdcall ADD(double* a,double* b,double* c)
{
   add_(a,b,c);
}
```

컴파일 방법

```
> g77 -c add.f
> gcc -c wrapG77toDVF.c
> df /c caller.f
> fl32 caller.obj add.o wrapG77toDVF.o
```

리스트 4. G77에서 컴파일된 포트란 루틴을 MSVC 및 GCC에서 호출하기

```
C add.f
      SUBROUTINE Add(A,B,C)
      DOUBLE PRECISION A,B,C

      C = A+B
      end

/* case1.c */
#include <stdio.h>

extern void add_(double* a,double* b,double*c);

int main()
{
    double a,b,c;
    a = 1.;
    b = 2.;
    add_(&a,&b,&c);
    printf("result : dd(1,2) = %f\n",c);

}
```

실행방법(G77→GCC)

```
> gcc case1.c add.f -lg2c
```

실행방법(G77→MSVC)

```
>  gcc -c add.f
>  cl case1.c add.o
```

이외에도 다양한 조합이 가능합니다. 물론 각각의 언어 및 컴파일러 조합에 따라 앞서에서 언급한 5가지를 고려해서 혼합컴파일을 수행해야 합니다. 다음 절은 지난번 연재에서 소개한 ATLAS, LAPACK, CHOLMOD 등을 여러 컴파일러 및 언어에서 호출하는 방법에 대해 알아보도록 하겠습니다.

2.2 ATLAS+LAPACK DLL 만들기

본격적으로 ATLAS+LAPACK를 활용하기에 앞서 지

리스트 5. 포트란 입출력문 제거

1. LAPACK/SRC/xerbla.f를 복사한 후 모두 comment 처리해 저장한다.
2. 다음과 같이 library를 수정한다.

```
$ g77-funroll-all-loops -O3 -c xerbla.f  또는 gcc
-funroll-all-loops -O3 -c xerbla.c
$ ar r liblapack.a xerbla.o
$ ar r libf77blas.a xerbla.o
```

리스트 6. ATLAS+LAPACK DLL 만들기

1. 다음 명령을 통해 GCC/G77용 DLL(blaslapack.dll)과 임포트 라이브러리(libblaslapack.a)를 생성한다. 만약 MinGW를 사용하면 -mno-cygwin 옵션이 필요없다.

```
> gcc  -shared -o blaslapack.dll
-Wl,--out-implib=libblaslapack.a
-Wl,--export-all-symbols
-Wl,--allow-multiple-definition
-Wl,--enable-auto-import -Wl,--whole-archive
liblapack.a libf77blas.a libcblas.a -Wl,-no-whole-archive
libatlas.a -lg2c
```

NOTE. 이 명령에서 blaslapack.dll과 임포트 라이브러리 libblaslapack.a가 생성된다. 위 명령은 msvcrt.dll을 사용하는 dll을 생성하게 된다. 만약 msvcr71에 링크되는 dll이 필요하다면 -lmsvcr71 옵션을 사용하고, gcc spec 파일(Cygwin\lib\gcc\i686-pc-cygwin\3.4.X or MinGW\lib\gcc\mingw32\3.4.X,)에서 -lmsvcr71로 변경해야 한다.

2. MSVC/DVF 용 임포트 라이브러리(blaslapeck.lib)는 다음과 같이 생성할 수 있다.

```
> pexports blaslapack.dll > blaslapack.def
> lib /machine:i386 /def:blaslapack.def
```

NOTE. DLL은 1번의 DLL을 그대로 쓰고, import library만 MSVC/DVF용으로 만듦. 그 이유는 MSVC/DVF에서 GCC용 임포트 라이브러리를 인식하기 못하기 때문이다.

난번 연재에서 생성한 라이브러리를 약간 수정하고, DLL로 만드는 작업을 먼저 수행하겠습니다. DLL은 동적연결 라이브러리(Dynamic Linked Library)를 의미하는데, 여러 프로그램에서 공용으로 사용되는 라이브러리의 경우 DLL형태로 만드는 경우가 많습니다. ATLAS+LAPACK은 수치해석을 위한 기본 라이브러리이기 때문에 DLL로 만드는 것이 그 성격에 맞는 것 같아, 정적라이브러리 및 동적라이브러리 두가지 형태로 활용하는 것을 다음 절에서 소개하겠습니다.

지난번 연재에서 생성한 ATLAS+LAPACK 관련 정적 라이브러는 liblapack.a, libcblas.a, libf77blas.a, libatlas.a 등 4개의 라이브러리 파일로 구성되어 있습니다. GCC 등에서 바로 이 라이브러리를 사용하는 것은 일부 포트란으로 작성된 오브젝트 파일에서 WRITE, READ 등 입출력문에 대한 컴파일러 의존성을 가지고 있기 때문에 MSVC, DVF 등에서 호출할 때 run-time library 오류가 발생할 수 있습니다. 리스트 5는 이와 같은 의존성을 제거하는 순서를 기술할 것입니다. ATLAS+LAPACK에서 READ, WRITE문을 가지는 포트란 파일은 xerbla.f에만 있기 때문에 이 파일에서 READ, WRITE문을 제거하는 것이 기본 아이디어입니다. 또 다른 방법으로른 C 언어로 대응되는 XERBLA로 만들어 대체할 수도 있습니다.

이제 수정된 라이브러리를 이용해 DLL로 만든 작업을 리스트 6에 제시하였습니다. pexports라는 유틸리티가 필요한데 인터넷에서 검색하면 쉽게 찾을 수 있습니다.

이상의 과정에서 ATLAS+LAPACK 관련 라이브러리를 정리하면 리스트 7과 같습니다.

리스트 7. ATLAS+LAPACK 라이브러리 요약

정적라이브러리
liblapack.a, libcblas.a, libf77blas.a, libatlas.a
동적라이브러리
blaslapack.dll, libblaslapack.a(GCC용 임포트 라이브러리), blaslapack.lib(MSVC용 임포트 라이브러리)

2.3 ATLAS+LAPACK 사용하기

여기에서 예제로 테스트할 함수는 BLAS의 DGEMV

리스트 8. DGEMV 및 DGESV의 인터페이스

1. DGEMV : Y = alpha*A*X + beta*y 또는 Y=alpha*A'*X + beta*Y
   - Fortran
     SUBROUTINE DGEMV(TRANS, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY)
     여기에서 TRANS는 CHARACTER*1으로서 'N'이면 A, 'T'면 A'를 사용
   - CBLAS
     void cblas_dgemv(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const int M, const int N, const double alpha, const double *A, const int lda, const double *X, const int incX, const double beta, double *Y, const int incY);
2. LAPACK 루틴 DGESV
   - Fortran
     SUBROUTINE DGESV(N, NRHS, A, LDA, IPIV, B, LDB, INFO)
   - 대응되는 C선언 :
     void dgesv_(int* N, int* NRHS, double* A, int* LDA, int* IPIV, double* B, int* LDB, int* INFO);

(cblas로는 cblas_dgemv)와 LAPACK의 DGESV입니다. 리스트 8에는 각 함수의 인터페이스를 요약하였습니다. BLAS 함수인 DGEMV는 C언어에 대한 interface 코드가 존재하지만 LAPACK 루틴은 존재하지 않습니다. 따라서 C언어에서 호출하기 위해서는 _DGESV를 함수 이름으로 사용해야 합니다.

(1) C에서 사용하기

리스트 9는 C용 테스트 프로그램 소스이고, 리스트 10과 11은 컴파일 및 링크 방법을 나타낸 것입니다.

리스트 9. C 테스트 프로그램

```
/* ctest.c */
#include <stdio.h>
#include "cblas.h"   /* for cblas_dgemv */
extern void dgesv_(int* N, int* NRHS, double* A, int*
LDA, int* IPIV, double* B, int* LDB, int* INFO );

int main()
{
  int i,j,N,NRHS,LDA,LDX,INFO;
  double ALPHA,BETA;
  double X[3] = {1.,3.,1.2};
  double AX[3];
  double A[9] = { 4.,  3.,  -1.,  3.,  5.,  0.,  -1.,  0.,  3.};
  double AA[9];
  int IPIV[3];

  printf("BLAS-LAPACK TEST\n");
  printf("IN A*X = B, A & B IS \n");
  for(i=0;i<3;i++)
  {
    for(j=0;j<3;j++)
      printf("%f ",A[i+j*3]);
    printf(" | %f\n",X[i]);
  }

  for(i=0;i<9;i++) AA[i] = A[i];

  N = 3;
  NRHS = 1;
  LDA = 3;
  LDX = 3;
  dgesv_(&N,&NRHS,A,&LDA,IPIV,X,&LDX,&INFO);
  printf("X= %f %f %f\n",X[0],X[1],X[2]);


  cblas_dgemv(CblasColMajor, CblasNoTrans,3, 3, 1, AA,3,
X,1,0.,AX,1);
  printf("A*X= %f %f %f\n",AX[0],AX[1],AX[2]);
  return 1;

}
```

리스트 10. GCC에서 사용하기

```
* static library 사용
> gcc -o ctest.exe ctest.c liblapack.a libcblas.a libatlas.a
> ctest.exe

* DLL 사용
> gcc -o ctest.exe ctest.c libblaslapack.a
> ctest.exe
```

리스트 11. MSVC에서 사용하기

```
* static library 사용
> cl ctest.c liblapack.a libcblas.a libatlas.a
> ctest.exe

* DLL 사용
> cl ctest.c blaslapack.lib
> ctest.exe
```

(2) 포트란에서 사용하기

리스트 12는 포트란용 테스트 프로그램 소스이고, 리스트 13은 G77에서 사용한 방법을 정리한 것입니다. DVF에

리스트 12. 포트란 테스트 프로그램

```
C FTEST.F
      PROGRAM TEST
      REAL*8 X(9),Y(3),A(3,3), AA(3,3),AX(3)
      REAL*8 ALPHA,BETA
      INTEGER IPIV(3),INFO

      X(1) = 1.
      X(2) = 3.
      X(3) = 1.2
      A(1,1) = 4.
      A(2,1) = 3.
      A(3,1) = -1.
      A(1,2) = 3.

      A(2,2) = 5.
      A(3,2) = 0.
      A(1,3) = -1.
      A(2,3) = 0.
      A(3,3) = 3.

      DO I=1,3
        DO J=1,3
         AA(I,J) = A(I,J)
        END DO
      END DO

      WRITE(*,*) 'BLAS-LAPACK TEST'
      WRITE(*,*) 'IN A*X = B, A & B IS '
      DO I=1,3
           WRITE(*,*) (A(I,J),J=1,3),'|', X(I)
      END DO

      CALL DGESV(3,1,A,3,IPIV,X,3,INFO)
      WRITE(*,*) 'X=',(X(I),I=1,3)

      CALL DGEMV(ICHAR('N'), 3, 3, 1.D0, AA, 3, X,
      1,0.D0,AX,1)
      WRITE(*,*) 'A*X = ',(AX(I),I=1,3)

      END
```

서 사용할 때는 Wrapper 코드가 필요한데 리스트 14에 나타냈으며, 사용하는 방법은 리스트 15에 정리하였습니다. 리스트 12에서 DGEMV를 호출할 때 첫 번째 인자로 'N'을 넘길 경우 G77은 이상없이 컴파일 되었으나, DVF에서는 문제가 발생하여 ICHAR() 함수를 이용해 ascii 코드 형태로 인자를 넘기도록 수정한 것입니다.

리스트 13. G77에서 사용하기

```
* static library 사용
> g77 -o ftest.exe ftest.f liblapack.a libcblas.a libf77blas.a
libatlas.a
> ftest.exe
 BLAS-LAPACK TEST
 IN A*X = B, A & B IS
  4.  3. -1.|  1.
  3.  5.  0.|  3.
 -1.  0.  3.|  1.20000005
 X= -0.214285706  0.728571423  0.328571447
 A*X =  1.  3.  1.20000005

* DLL 사용
> g77 -o ftest.exe ftest.f libblaslapack.a
> ftest.exe
```

리스트 14. DVF용 Wrapper 코드

```
/* wrap4DVF.c */
extern void dgemv_(unsigned char* TRANS, int* M, int*
                   N, double* ALPHA, double* A, int*
                   LDA, double* X, int* INCX,  double*
                   BETA, double* Y, int* INCY );

void __stdcall DGEMV(unsigned char* TRANS, int* M,
                     int* N, double* ALPHA, double*
                     A, int* LDA, double* X, int*
                     INCX, double* BETA, double* Y,
                     int* INCY )
{
  dgemv_(TRANS, M, N, ALPHA, A, LDA, X, INCX,
         BETA, Y, INCY);
}

extern void dgesv_(int* N, int* NRHS, double* A, int*
                   LDA, int* IPIV, double* B, int* LDB,
                   int* INFO );

void __stdcall DGESV(int* N, int* NRHS, double* A, int*
                     LDA, int* IPIV, double* B, int*
                     LDB, int* INFO )
{
   dgesv_( N, NRHS, A, LDA, IPIV, B, LDB, INFO );
}
```

리스트 15. DVF에서 사용하기

```
* static library 사용
> gcc -c wrap4DVF.c
> df -c ftest.f
> fl32 ftest.obj wrap4DVF.o liblapack.a libcblas.a libatlas.a
libf77blas.a
> ftest.exe

* DLL 사용
> gcc -c wrap4DVF.c
> df -c ftest.f
> fl32 ftest.obj wrap4DVF.o blaslapack.lib
> ftest.exe
```

2.4 CHOLMOD 사용하기

이제 CHOLMOD를 C/C++와 포트란에서 활용하는 방법에 대해 설명하기로 하겠습니다. 먼저 CHOLMOD를 사용하는데 필요한 라이브러리는 이전 절의 ATLAS+LAPACK 라이브러리와 지난번 연재에서 작성한 libcholmod.a libamd.a libcolamd.a libccolamd.a libmethis.a 등이 필요합니다. CHOLMOD를 사용하기 위해서는 CSC format(Compressed Sparse Column format)이라는 희소행렬을 저장하는 포맷을 이해해야 합니다. 이 부분은 지난번 연재를 참조하거나 CHOLMOD 매뉴얼을 참조하시기 바랍니다. 리스트 16은 CHOLMOD에서 사용되는 주요함수를 나열한 것입니다. CHOLMOD를 사용하기 위해서는 먼저 cholmod_start로 사용할 준비를 하고, symbolic analysis, factorization, solve 과정을 거쳐 해를 구하게 됩니다. CHOLMOD는 메모리 관리를 직접하기 때문에 cholmod_allocate로 시작하는 자체 메모리 할당 함수가 있습니다. 따라서 이 함수들을 이용해서 행렬이나 벡터에 대한 메모리를 할당합니다. 자세한 설명은 매뉴얼을 참조하기 바란다.

리스트 16. CHOLMOD의 주요함수

```
int cholmod_start(cholmod_common *Common) ;
int cholmod_finish(cholmod_common *Common) ;
cholmod_factor    *cholmod_analyze(cholmod_sparse    *A,
cholmod_common *Common);
int cholmod_factorize(cholmod_sparse *A, cholmod_factor
*L, cholmod_common *Common) ;
cholmod_dense *cholmod_solve(int sys, cholmod_factor *L,
                             cholmod_dense *B, cholmod_
                             common Common) ;
cholmod_sparse *cholmod_allocate_sparse(size_t nrow, size_t
                ncol, size_t nzmax, int sorted, int packed, int stype,
                int xtype, cholmod_common *Common) ;
int cholmod_free_sparse(cholmod_sparse
**A,cholmod_common *Common) ;
cholmod_dense *cholmod_allocate_dense
    (size_t nrow,size_t ncol,size_t d,int xtype, cholmod_common
    *Common) ;
int cholmod_free_dense(cholmod_dense **X,
    cholmod_common *Common) ;
int cholmod_print_sparse(cholmod_sparse *A, char
    *name,cholmod_common *Common) ;
```

(1) C/C++에서서의 호출

리스트 17은 4×4 대칭 행렬을 CHOLMOD를 이용해 해

를 구하는 과정을 나타낸 예제이고, 리스트 18은 컴파일 및 링크과정을 나타낸 것입니다.

리스트 17. CHOLMOD 사용예

```
/* cholCTest.c */

/*
[ 1.0 0.5 0.0 0.0 ]        [ 1.0 ]
[ 0.5 1.0 0.5 0.0 ]        [ 2.0 ]
[ 0.0 0.5 1.0 0.5 ] X   = [ 3.0 ]
[ 0.0 0.0 0.5 1.0 ]        [ 4.0 ]
x = [ 0 2 0 4]'
*/

#include "./cholmod/cholmod.h"
#include "stdio.h"

#define TRUE 1
#define FALSE 0

int main (void)
{
    double one [2] = {1,0}, m1 [2] = {-1,0} ;
                                    /* basic scalars */
    cholmod_common c;
    int nrow=4;
    int ncol=4;
    int nzmax=7;
    cholmod_sparse *A;
    int* pcol;
    int* irow;
    double* val;
    cholmod_dense* b;
    double* bp;
    cholmod_factor *L;
    int ret;
    cholmod_dense *x,*r;

    cholmod_start (&c) ;    /* start CHOLMOD */

    A = cholmod_allocate_sparse(nrow, ncol, nzmax, 1,
        1, -1, CHOLMOD_REAL,&c);
    c.print = 4;  // full print

    if(A == NULL)
    {
      cholmod_free_sparse(&A,&c);
      cholmod_finish(&c);
    }
    pcol = (int*) A->p;
    irow = (int*) A->i;  // this is void pointer.
    val = (double*) A->x;

    pcol[0]=0; pcol[1]=2; pcol[2]=4; pcol[3]=6; pcol[4]=7;
    irow[0]=0; irow[1]=1; irow[2]=1; irow[3]=2;
    irow[4]=2; irow[5]=3; irow[6]=3;
    val[0]=1.; val[1]=0.5; val[2]=1.; val[3]=0.5; val[4]=1.;
    val[5]=0.5; val[6]=1.0;

    cholmod_print_sparse (A, "A", &c) ;
                                    /* print the matrix */
    /* b = ones(n,1) */
    b = cholmod_allocate_dense (A->nrow, 1, A->nrow,
        A->xtype, &c) ;
```

리스트 17. CHOLMOD 사용예(계속)

```
    bp = (double*)b->x;
    bp[0] = 1.; bp[1] = 2.; bp[2] = 3.; bp[3] = 4.;

    L = cholmod_analyze (A, &c) ;    /* analyze. if error,
        L is freed automatically */
    if (L == NULL) // or c.status < CHOLMOD_OK
    {
        printf("ERROR");
        return 1;
    }

    ret = cholmod_factorize(A, L, &c) ;    /* factorize */
    if (ret == FALSE) // or c.status < CHOLMOD_OK)  // error
    {
      printf("ERROR");
      return 1;
    }

    x = cholmod_solve (CHOLMOD_A, L, b, &c) ;
                                    /* solve Ax=b */
    cholmod_print_dense(x,"x",&c);

    r = cholmod_copy_dense (b, &c) ;    /* r = b */
    cholmod_sdmult (A, 0, m1, one, x, r, &c) ;    /* r
= r-Ax */
    printf ("norm(b-Ax) %8.1e\n",cholmod_norm_dense (r, 0, &c)) ;
                                    /* print norm(r) */

    cholmod_free_factor (&L, &c) ;  /* free matrices */
    cholmod_free_sparse (&A, &c) ;
    cholmod_free_dense (&x, &c) ;
    cholmod_free_dense (&b, &c) ;
    cholmod_finish (&c) ;   /* finish CHOLMOD */
    return 0 ;
}
```

리스트 18. C에서의 컴파일 과정

* MinGW GCC + BLAS/LAPACK DLL사용

```
> gcc -o cholCTest.exe cholCTest.c libcholmod.a libamd.a
libcolamd.a libccolamd.a libmetis.a libblaslapack.a
> cholTest.exe
```

* MinGW GCC + BLAS/LAPACK 정적 라이브러리 사용

```
> gcc -o cholCTest.exe cholCTest.c libcholmod.a libamd.a
libcolamd.a libccolamd.a libmetis.a liblapack.a libcblas.a
libf77blas.a libatlas.a
> cholCTest.exe
```

* MSVC+ BLAS/LAPACK DLL사용

```
> cl cholCTest.c libcholmod.a libamd.a libcolamd.a libccolamd.a
libmetis.a blaslapack.lib
> cholCTest.exe
```

* MSVC + BLAS/LAPACK 정적 라이브러리 사용

```
> cl cholCTest.c libcholmod.a libamd.a libcolamd.a libccolamd.a
libmetis.a liblapack.a libcblas.a libf77blas.a libatlas.a
> cholCTest.exe
```

NOTE 1. MinGW GCC를 사용할 경우 라이브러리의 기입순서에 주의해야 한다. 호출하는 쪽의 라이브러리가 호출 당하는 라이브러리보다 먼저 와야 한다.

NOTE 2. MSVC를 사용할 경우 library의 확장자가 .a로 끝나기 때문에 경고메세지가 발생한다. 이를 해결하기 위해서는 모두 .lib으로 만들면 된다.

(2) Fortran에서의 사용법

CHOLMOD는 C 라이브러리로서 기본적으로 포트란을 지원하지 않습니다. 포트란에서 사용하기 위해 리스트 19

리스트 19. CHOLMOD용 interface 코드

```
/* cholsol.c */
#include "./cholmod/cholmod.h"
#include <stdio.h>

#define TRUE 1
#define FALSE 0

/* interface code for CHOLMODE. The routine acting like

   SUBROUTINE CHOLMOD(X, N, ITYPE, VAL,
   ICOLPTR, IROW, B, INFO)

   for solving A*X = B. It is a kind of driver routine.

   X(N) : solution vector(output)
   N    : matrix size
   ITYPE : CSC matrix type, 1 for upper triangle, -1 for
           lower triangle
   VAL(NZMAX), ICOLPTR(N+1), IROW(NZMAX) :
           Symtem matrix (CSC format matrix)
                   Fortran-style 1-based array is used.
   B(N) : right-hand side vector
   INFO : output information
          1 for sucess, 1 for memory allocation error, 2 for
          analyze error,
         3 for factorization error
*/

#ifdef DVF
void __stdcall CHOLSOL(double* x,int* n,int*
itype,double* val,int* pcol,int* irow,double* b,int* info)
#else /* G77 */
void cholsol_(double* x,int* n,int* itype,double* val,int*
pcol,int* irow,double* b,int* info)
#endif
{
    cholmod_common c;
    int nrow=*n;
    int ncol=*n;

    int nzmax;
    cholmod_sparse *A;
    cholmod_dense *B,*X;
    cholmod_factor *L;
    int* Ap;
    int* Ai;
    double* Av;
    double* Bp;
    double* Xp;
    int i,ret;
    nzmax= pcol[*n]-1;  /* make 0-based */
```

리스트 19. CHOLMOD용 interface 코드(계속)

```
    cholmod_start (&c) ;            /* start CHOLMOD */

    A = cholmod_allocate_sparse(nrow,ncol,nzmax,1,1,itype,
        CHOLMOD_REAL, &c);
    if(A == NULL)
    {
      cholmod_free_sparse(&A,&c);
      cholmod_finish(&c);
      *info = 1; // memory allocation error
       return;
    }
    Ap = (int*) A->p;
    Ai = (int*) A->i;  // this is void pointer.
    Av = (double*) A->x;

    for(i=0;i<=nrow;i++) Ap[i] = pcol[i]-1;  /* make 0-based */
    for(i=0;i<nzmax;i++)
    {
       Ai[i] = irow[i]-1;    /* make 0-based */
       Av[i] = val[i];
    }
    B = cholmod_allocate_dense (A->nrow, 1, A->nrow,
         A->xtype, &c) ;
    Bp = (double*)B->x;
    for(i=0;i<nrow;i++) Bp[i] = b[i];

    L = cholmod_analyze (A, &c) ;     /* analyze. if error,
        L is freed automatically */
    if (L == NULL) /* or c.status < CHOLMOD_OK */
    {
       *info = 2; /* analyze error */
       return;
    }
   ret = cholmod_factorize(A, L, &c) ;     /* factorize */

    if (ret == FALSE) /* or c.status < CHOLMOD_OK)  */
    {
       *info = 3; // factorize error
       return;
    }

    X = cholmod_solve (CHOLMOD_A, L, B, &c) ;
/* solve Ax=b */
    Xp = (double*)X->x;

    for(i=0;i<nrow;i++) x[i] = Xp[i];

    cholmod_free_factor (&L, &c) ;     /* free matrices */
    cholmod_free_sparse (&A, &c) ;
    cholmod_free_dense (&X, &c) ;
    cholmod_free_dense (&B, &c) ;
    cholmod_finish (&c) ;           /* finish CHOLMOD */
    *info = 0;
}
```

와 같이 Ax = b를 푸는 드라이버 루틴 CHOLSOL을 C로 작성하여 포트란에서 불러 쓰도록 하였습니다. 이러한 드라이버 루틴을 필요에 따라 작성해야 포트란에서 사용할 수 있습니다. 리스트 20과 리스트 21은 테스트 프로그램 소스와 컴파일 및 링크과정입니다.

리스트 20. CHOLSOL 테스트 프로그램

```
c234567 cholFTest.f
c
c [ 1.0 0.5 0.0 0.0 ]       [ 1.0 ]
c [ 0.5 1.0 0.5 0.0 ]       [ 2.0 ]
c [ 0.0 0.5 1.0 0.5 ] X   = [ 3.0 ]
c [ 0.0 0.0 0.5 1.0 ]       [ 4.0 ]
c x = [ 0 2 0 4]'
      PROGRAM CHOLFTEST
      IMPLICIT NONE
      INTEGER N,I,NZMAX,INFO
      PARAMETER (N=4,NZMAX=7)
      DOUBLE PRECISION VAL(NZMAX),B(N), X(N)
      INTEGER ICOLPTR(N+1), IROW(NZMAX)

      DATA ICOLPTR/1,3,5,7,8/, IROW/1,2,2,3,3,4,4/
      DATA VAL/1.0,0.5,1.0,0.5,1.0,0.50,1.0/
      DATA B/1.0,2.0,3.0,4.0/

      WRITE(*,*) 'ICOLPTR = ',(ICOLPTR(I),I=1,N+1)
      WRITE(*,*) 'IROW = ',(IROW(I),I=1,NZMAX)

      WRITE(*,*) 'A = ', (VAL(I),I=1,NZMAX)
      WRITE(*,*) 'B = ', (B(I),I=1,N)

      CALL CHOLSOL(X,4,-1,VAL,ICOLPTR,IROW,B,INFO)
      IF(INFO .EQ. 0) THEN
        WRITE(*,*) 'SUCCESS'
      ELSE IF(INFO .EQ. 1) THEN
        WRITE(*,*) 'MEMORY ALLOCATION ERROR'
      ELSE IF(INFO .EQ. 2) THEN
        WRITE(*,*) 'ANALYZE ERROR'
      ELSE IF(INFO .EQ. 3) THEN
        WRITE(*,*) 'FACTORIZATION ERROR'
      END IF

      WRITE(*,*) 'X = ',(X(I),I=1,4)

      END
```

리스트 21. 포트란에서 호출하는 방법

```
* MinGW G77 + BLAS/LAPACK DLL사용
> g77 -o cholFTest.exe cholFTest.f cholsol.c libcholmod.a
libamd.a libcolamd.a libccolamd.a libmetis.a libblaslapack.a
> cholTest.exe

* MinGW G77 + BLAS/LAPACK 정적 라이브러리 사용
> g77 -o cholFTest.exe cholFTest.f cholsol.c libcholmod.a
libamd.a libcolamd.a libccolamd.a libmetis.a liblapack.a
libcblas.a libf77blas.a libatlas.a
> cholCTest.exe

* DVF + BLAS/LAPACK DLL사용
> gcc -c -DDVF cholsol.c
> df /c cholFTest.f
> fl32 cholFTest.obj cholsol.o libcholmod.a libamd.a
libcolamd.a libccolamd.a libmetis.a blaslapack.lib
> cholFTest.exe
```

리스트 21. 포트란에서 호출하는 방법(계속)

```
* DVF + BLAS/LAPACK 정적 라이브러리 사용
> gcc -c -DDVF cholsol.c
> df /c cholFTest.f
> fl32 cholFTest.obj cholsol.o libcholmod.a libamd.a
libcolamd.a libccolamd.a libmetis.a liblapack.a libcblas.a
libf77blas.a libatlas.a
> cholFTest.exe
```

3. 스크립트 언어(Script Language)

스크립트 언어는 소스 파일로부터 실행파일(.exe)를 만들지 않고 행 단위로 실행되는 언어를 의미합니다. 최근 많이 사용되는 스크립트 언어로는 Python, Tcl/Tk, Perl 등이 있습니다. 스크립트 언어의 장점은 사용하기가 편리하고, 보통은 free software로 배포되며, 다른 프로그램에 붙여 사용하기가 편리하다는 점입니다. 최근에는 이 스크립트 언어를 유한요소 패키지에 붙여 사용하는 경우가 늘고 있습니다. 예를 들어 ABAQUS에서는 Python을 사용하고, OpenSees는 Tcl/Tk를 사용합니다. 여기에서는 제가 주로 사용하는 언어인 Python에 대해서 간략하게 언급

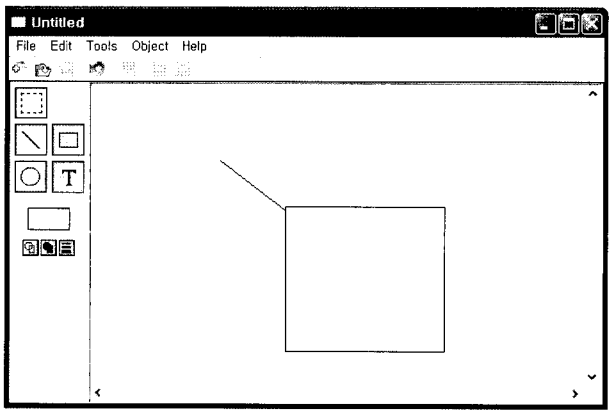

그림 1. Python으로 작성된 GUI 프로그램(wxPython 이용)

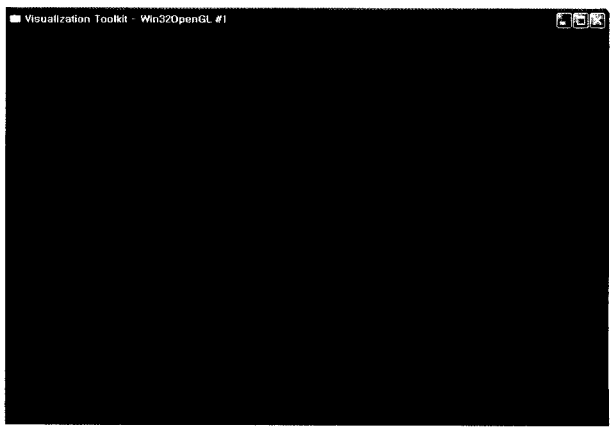

그림 2. VTK를 이용한 유한요소메쉬의 표현

하는 것으로 마무리하겠습니다.

Python은 Perl과 함께 객체지향언어를 지향하는 스크립트 언어입니다. www.python.org에서 배포본을 다운로드할 수 있으며, 각종 그래픽, 수치라이브러리를 추가로 인스톨 해서 사용할 수 있습니다. 이러한 라이브러리는 마치 Matlab에서 툴박스를 깔아쓰는 것과 같이 사용할 수 있습니다. Python은 가장 큰 장점은 빠른 프로그램 개발이 가능하다는 점입니다. 예를 들어 최근 많이 쓰는 GUI용 라이브러리인 wxPython을 이용할 경우 간단한 GUI 인터페이스를 가진 프로그램을 금방 완성할 수 있습니다. 또한, 어떤 라이브러리를 이해하는 데 있어 행단위의 실행으로 그 라이브러리를 쉽게 이해할 수 있도록 해준다는 장점이 있습니다. 다음 연재에서 소개할 3차원 시각화 라이브러리인 VTK(Visualization Toolkit)의 경우 Python을 이용해 사용하는 방법을 습득하는 것이 좋습니다.

4. 맺음말

이번 연재에서는 혼합컴파일 기법에 초점을 맞추어 상당히 기술적인 부분에 대해 설명하였습니다. 그리고 간단하게 스크립트 언어를 소개하였습니다. 다음 연재의 주제는 Visualization로써 VTK(Visualization Toolkit)이라는 3차원 시각화 라이브러리(물론 free library입니다)를 이용한 유한요소법에서 필요한 시각화 기법을 소개합니다.