Chapter 9

Providing Mixed-Language and Legacy Support in a Library: Experiences of Developing PETSc

Satish Balay, Jed Brown, Matthew Knepley, Lois Curfman McInnes, and Barry Smith

9.1 Introduction

This chapter explains how numerical libraries written in C can portably support its use from both modern and legacy versions of Fortran efficiently. This is done by examining, in a particular library, all the cross-language issues in mixing C and Fortran. Despite the chagrin of many computer scientists, scientists and engineers continue to use Fortran to develop new simulation codes, and the Fortran language continues to evolve with new standards and updated compilers. The need to combine Fortran and C code will also continue, therefore, will be no less important in future computing systems that include many-core processing with a hierarchy of memories and the integration of GPU systems with CPU systems all the way up to exascale systems. Thus, numerical analysts and other developers of mathematical software libraries must ensure that such libraries are usable from Fortran. To make the situation more complicated, depending on the age of the Fortran application (or the age of its developers), the Fortran source code may be Fortran 77, Fortran 90, Fortran 2003, or Fortran 2008 (possibly with TS 29113, required by MPI-3's "mpi_f08" module). In fact, the same Fortran application may include source files with the suffix .f that utilize Fortran 77 syntax and formatting (traditional fixed format), .F files that utilize some Fortran 90 or later language features but still have fixed format, and .F90 that use free format. Many Fortran application developers also resist utilizing the more advanced features of recent Fortran standards for a variety of reasons. Thus, any interlanguage Fortran library support must support both traditional Fortran dialects and modern features such as derived types. See [325] for a short history of Fortran.

The Babel project [321] was an ambitious effort to develop a language-independent object model that would allow scientific software libraries written in several languages to be utilized from any of the languages, much as Corba [326] was for business software. However, because of its extremely ambitious nature, the tools (Java) selected to develop the model, and insufficent funding for the large amount of development needed, the software could not fully serve application and library needs.

The Portable Extensible Toolkit for Scientific computation (PETSc) is a portable C software library for the scalable solution of linear, nonlinear, and ODE/DAE systems, and computation of adjoints (sometimes called sensitivities) of ODE systems. PETSc has been developed and supported at ANL for the past 20 years. PETSc has always supported a uniform Fortran interface [319], even in the very early phases of library development (see page 29 of) [323], [324]. PETSc is written using a C-based object model (in fact, that model inspired the Babel design) with a mapping of the objects and methods (functions) on the objects to Fortran as well as Python. This paper discussses only the Fortran mapping in PETSc.

9.2 Fortran-C Interfacing Issues and Techniques

Prior to the development of Fortran 2003, there was no standard within Fortran for interfacing with C code. That is, how can one call Fortran subroutines from C and C functions from Fortran in portable code as efficiently as possible. This means not needing to copy entire data structures between the two languages. Fortunately, since all Fortran compilers followed the same general set of protocols, one has always been able to portably mix Fortran and C code. The interlanguage issues that must be dealt with include the following.

Symbol names: Fortran compilers convert the symbol names to all capitals, all lower case, or all lower case with an underscore suffix. One variant is that symbols with an underscore get an additional underscore at the end of the symbol. In PETSc we handle this name mangling using the preprocessor, with code such as

```
#if defined(PETSC_HAVE_FORTRAN_CAPS)
#define matcreateseqaij_   MATCREATESEQAIJ
#elif !defined(PETSC_HAVE_FORTRAN_UNDERSCORE)
```

```
#define matcreateseqaij_   matcreateseqaij
endif
```

A terser, arguably better way of managing this is to use the paste ## feature of the C preprocessor. First we define the macro `FC_FUNC()` based on the Fortran symbol format.

```
#if defined(PETSC_HAVE_FORTRAN_CAPS)
#define FC_FUNC(name,NAME) NAME ## _
#elif !defined(PETSC_HAVE_FORTRAN_UNDERSCORE)
#define FC_FUNC(name,NAME) name
#else
#define FC_FUNC(name,NAME) name ## _
#endif
```

Defining each symbol then takes only a single line, such as

```
#define matcreateseqaij_ FC_FUNC(matcreateseqaij,MATCREATESEQAIJ)
```

Character strings: Since Fortran strings are not null terminated, the Fortran compiler must generate additional code to indicate the length of each string. Most Fortran compilers include the string length (as an integer) as an additional argument at the end of the calling sequence; some compilers pass the length (as an integer) immediately after the character argument. In PETSc we handle this issue in the definition of our C stub function, again using the preprocessor `#define`, with code such as

```
void STDCALL vecsetoptionsprefix_(Vec *v,char* prefix
                                  PETSC_MIXED_LEN(len),
                                  PetscErrorCode *ierr
                                  PETSC_END_LEN(len))
char *t;
FIXCHAR(prefix,len,t);
*ierr = VecSetOptionsPrefix(*v,t);
FREECHAR(prefix,t);
```

Here

```
#define FIXCHAR(a,n,b)
{
  if (a == PETSC_NULL_CHARACTER_Fortran) {
    b = a = 0;
  } else {
    while((n > 0) && (a[n-1] == ' ')) n--;
    *ierr = PetscMalloc((n+1)*sizeof(char),&b);
    if (*ierr) return;
    *ierr = PetscStrncpy(b,a,n+1);
    if (*ierr) return;
  }
}
```

allocates a null terminated version of the string to pass to C and

```
#define FREECHAR(a,b) if (a != b) PetscFreeVoid(b);
```

frees the temporary string. Depending on where the Fortran compiler places the `len` argument, either the `PETSC_MIXED_LEN(len)` or the `PETSC_END_-LEN(len)` macro simply removes the argument.

Stack management: Microsoft (and compilers for Microsoft systems) supports various ways that function arguments are pushed on the stack and if the caller or the callee removes the stack frame. Microsoft provides macros (used in the function prototypes in C) to indicate the convention that each function is using. Early Fortran compilers for Microsoft used the __stdcall call convention, while C and C++ did not; hence, C prototypes for each Fortran function required this annotation. Modern Intel Fortran compilers on Microsoft systems do not use the __stdcall call convention, and hence this annotation is needed only for older Fortran compilers on Microsoft systems. In PETSc we handle this issue by decorating the C function definition with STDCALL that becomes __stdcall only with older Fortran compilers for Microsoft systems.

Include files: Although the Fortran 77 standard did not provide for include files, most Fortran compilers support include files that use the C preprocessor (CPP) syntax; and for those systems that do not, one can always call the C preprocessor on the Fortran source and then call the Fortran compiler on the result. The use of include files with Fortran code makes possible many of the techniques utilized by PETSc (and discussed below). Full C/Fortran interoperability can be provided without requiring the use of Fortran include files, instead, for example, utilizing Fortran modules to contain the needed common data and values.

Enums: Since Fortran 77 provided no concept of the C enum, the established practice was to use

```
#define PetscEnum  integer
```

and then define each enum type with, for example,

```
#define InsertMode PetscEnum
```

The enumerated values are set by using the Fortran parameter statement, for example,

```
      PetscEnum INSERT_VALUES
      parameter (INSERT_VALUES=1)
      PetscEnum ADD_VALUES
      parameter (ADD_VALUES=2)
```

although care must be taken that the same integer values are used in the C and Fortran code. Recent versions of Fortran support enums via

```
ENUM  InsertMode
      ENUMERATOR :: INSERT_VALUE
      ENUMERATOR :: ADD_VALUE
   END ENUM
ENUM, BIND(C) :: InsertMode
```

which automatically ensures that the values assigned in Fortran match those in the C enum.

Compile and runtime constants: As demonstrated above, the Fortran parameter statement can be used to set compile-time constants that must match between C and Fortran. But what about runtime constants? These are traditionally handled by using common blocks whose entries are initialized from the C values when the package is initialized. For example, here is how PETSc handles the runtime value PETSC_COMM_WORLD. In a Fortran include file we define

```
      MPI_Comm PETSC_COMM_WORLD
      common /petscfortran/ PETSC_COMM_WORLD
```

Then after PETSC_COMM_WORLD is defined in C within `PetscInitialize()`, it calls the Fortran routine

```
     subroutine PetscSetCommonBlock(c1)
      implicit none
#include      <petsc/finclude/petscsys.h>
      integer c1
      PETSC_COMM_WORLD    = c1
      return
      end
```

to store the value in a Fortran common block. If one is willing to give up portability to pure Fortran 77 codes, then these values can be stored in a module rather than a common block. Initial communicating of Fortran runtime constants to C is handled similarly except that the Fortran code calls C with the required values. For example,

```
     subroutine PetscSetFromCommonBlock()
      implicit none
#include      <petsc/finclude/petscsys.h>
```

```
      call PetscSetFortranBasePointers(PETSC_NULL_CHARACTER,
         PETSC_NULL_INTEGER,PETSC_NULL_DOUBLE,PETSC_NULL_OBJECT,
         PETSC_NULL_FUNCTION)
      return
      end
```

passes the addresses within a common block to C.

```
      PetscChar(80)       PETSC_NULL_CHARACTER
      PetscInt            PETSC_NULL_INTEGER
      PetscFortranDouble  PETSC_NULL_DOUBLE
      PetscObject         PETSC_NULL_OBJECT
      external PETSC_NULL_FUNCTION
      common /petscfortran/ PETSC_NULL_CHARACTER,
                            PETSC_NULL_INTEGER,
                            PETSC_NULL_DOUBLE,
                            PETSC_NULL_OBJECT
```

The following C routine called from Fortran then puts the values of the Fortran common block addresses and external function into global C variables.

```
void STDCALL petscsetfortranbasepointers_(char *fnull_character
                                       PETSC_MIXED_LEN(len),
                                       void *fnull_integer,
                                       void *fnull_double,
                                       void *fnull_object,
                                       void (*fnull_func)(void)
                                       PETSC_END_LEN(len))
{
  PETSC_NULL_CHARACTER_Fortran = fnull_character;
  PETSC_NULL_INTEGER_Fortran   = fnull_integer;
  PETSC_NULL_DOUBLE_Fortran    = fnull_double;
  PETSC_NULL_OBJECT_Fortran    = fnull_object;
  PETSC_NULL_FUNCTION_Fortran  = fnull_func;
}
```

Note that since traditional Fortran has no concept of a common block variable declared as a function pointer, the `PETSC_NULL_FUNCTION` is simply declared with the `external` marker. This construct for managing null pointer usage in Fortran is needed because Fortran has no concept of a generic `NULL`. Instead, one needs a `NULL` for each data type; then in the C stub called from Fortran, the specific `NULL` data type is converted to the C `NULL`, for example,

```
void STDCALL matcreateseqaij_(MPI_Comm *comm,PetscInt *m,
                              PetscInt *n,PetscInt *nz,
                              PetscInt *nnz,Mat *newmat,
                              PetscErrorCode *ierr)
```

```
{
  if ((void*)(uintptr_t)nnz == PETSC_NULL_INTEGER_Fortran)
    { nnz = NULL; }
  *ierr = MatCreateSeqAIJ(MPI_Comm_f2c(*(MPI_Fint*)comm),*m,*n,
                                                  *nz,nnz,newmat);
}
```

PETSc also has many runtime constants in the style of `MPI_COMM_WORLD`, such as `PETSC_VIEWER_STDOUT_WORLD`, which are handled similarly but are compile-time constants in Fortran. In Fortran they are defined as integers via the parameter statement.

```
     PetscFortranAddr PETSC_VIEWER_STDOUT_WORLD
     parameter (PETSC_VIEWER_STDOUT_WORLD = 8)
```

Then the C stub checks whether the input is one of these special values and converts to the appropriate runtime C value, for example,

```
#define PetscPatchDefaultViewers_Fortran(vin,v)
{
  if ((*(PetscFortranAddr*)vin) ==
                                 PETSC_VIEWER_DRAW_WORLD_FORTRAN){
    v = PETSC_VIEWER_DRAW_WORLD;
  } else if ((*(PetscFortranAddr*)vin) ==
                                 PETSC_VIEWER_DRAW_SELF_FORTRAN){
    v = PETSC_VIEWER_DRAW_SELF;
  } else if ((*(PetscFortranAddr*)vin) ==
                                PETSC_VIEWER_STDOUT_WORLD_FORTRAN){
    v = PETSC_VIEWER_STDOUT_WORLD;
  } else ...
  } else {
    v = *vin;
  }
}
void STDCALL vecview_(Vec *x,PetscViewer *vin,PetscErrorCode *ierr)
{
  PetscViewer v;
  PetscPatchDefaultViewers_Fortran(vin,v);
  *ierr = VecView(*x,v);
}
```

Pointers in traditional Fortran to C arrays: PETSc makes widespread use of array pointers in its API to allow efficient programmers access to "raw" data structures. For example,

```
double *x;
Vec v;
VecGetArray(v,&x);
```

gives users direct access to the local values with a vector. Traditional Fortran has no concept of an array pointer, which would severely limit the use of some of PETSc's functionality from traditional Fortran. Fortunately, again, despite there having been no Fortran standard for this type of functionality, it is still achievable and has been used in PETSc for over 20 years. In the user's Fortran code, an array of size one is declared as well as an integer long enough to access anywhere in the memory space from that array offset (`PetscOffset` is a 32-bit integer for 32-bit memory systems and a 64-bit integer for 64-bit memory systems).

```
Vec X
PetscOffset  lx_i
PetscScalar lx_v(1);
```

They then call, for example,

```
call VecGetArray(X,lx_v,lx_i,ierr)
call InitialGuessLocal(lx_v(lx_i),ierr)
call VecRestoreArray(X,lx_v,lx_i,ierr)
```

where `InitialGuessLocal` is defined, for example, as

```
subroutine InitialGuessLocal(x,ierr)
implicit none
PetscInt xs,xe,ys,ye
common /pdata/  xs,xe,ys,ye
PetscScalar    x(xs:xe,ys:ye)
PetscErrorCode ierr

!  compute entries in the local portion of x()
```

This approach uses the fact that traditional Fortran passes by pointer, as opposed to by value, and when passing array pointers does not distinguish between one and multidimensional arrays. The PETSc C routine that manages this is similar to the following code

```
void STDCALL vecgetarray_(Vec *x,PetscScalar *fa,size_t *ia,
                                        PetscErrorCode *ierr)
{
  PetscScalar *lx;
  *ierr = VecGetArray(*x,&lx); if (*ierr) return;
  *ierr = PetscScalarAddressToFortran(fa,lx,ia);
}
```

where

```
PetscErrorCode PetscScalarAddressToFortran(PetscScalar *base,
                                           PetscScalar *addr,
                                                    size_t *ia)
{
  size_t   tmp1 = (size_t) base,tmp2;
  size_t   tmp3 = (size_t) addr;

  if (tmp3 > tmp1) {/* C address is larger than Fortran
    address */
    tmp2 = (tmp3 - tmp1)/sizeof(PetscScalar);
    *ia  =  tmp2;
  } else {  /* Fortran address is larger than C address */
    tmp2 = (tmp1 - tmp3)/sizeof(PetscScalar);
    *ia  = -((size_t) tmp2);
  }
}
```

calculates the appropriate signed displacement between the Fortran (dummy) array and the actual C array. Although this may seem like a dangerous "pointer" trick, it has worked for over 20 years on all systems to which we have access. One caveat is that if the Fortran compiler contains support for array "out of bounds" checking, this feature must be turned off (for example, the IBM Fortran compiler has a command line option to turn on this checking).

Array pointers in F90 to C arrays: With Fortran 90 array pointers it became possible to simplify the Fortran user interface for routines such as `VecGetArray()` to

```
PetscScalar,pointer :: lx_v(:)
Vec X

call VecGetArrayF90(X,lx_v,ierr)
```

This is implemented in PETSc by the C stub

```
void STDCALL vecgetarrayf90_(Vec *x, F90Array1d *ptr,int *ierr
                             PETSC_F90_2PTR_PROTO(ptrd))
{
  PetscScalar *fa;
  PetscInt    len,one = 1;
  *ierr = VecGetArray(*x,&fa);      if (*ierr) return;
  *ierr = VecGetLocalSize(*x,&len); if (*ierr) return;
  *ierr = F90Array1dCreateScalar(fa,&one,&len,ptr
                                 PETSC_F90_2PTR_PARAM(ptrd));
}
```

that calls the Fortran routine

```
subroutine F90Array1dCreateScalar(array,start,len1,ptr)
```

```
implicit none
#include <petsc/finclude/petscsys.h>
PetscInt start,len1
PetscScalar, target ::  array(start:start+len1-1)
PetscScalar, pointer :: ptr(:)
ptr => array
end subroutine
```

The Portland Group Fortran compiler passes additional information about each of the Fortran pointer arrays through final (hidden) arguments to the called functions. With this system the `PETSC_F90_2PTR_PROTO(ptrd)` is defined; on all other systems it generates nothing. The same general mechanism outlined above for PetscScalar one-dimensional arrays also works (with modification) for multiple-dimensional arrays as well as arrays of integers. One would think that with support for using F90 array features there would be no need to continue to support the F77 compatible `VecGetArray()`; yet, surprisingly large numbers of PETSc users continue to use the older version.

Portable Fortran source and include files: The Fortran standards provide a file format that is safe to use for all Fortran standards. This format uses exclusively the ! in the first column, only numerical values in the second to fifth column, a possible continuation character of & in the sixth column, Fortran commands in the seventh to 71st column, and a possible continuation character of & after the 72nd column. As long as this formatting is obeyed in the libraries' include files and source code, the code will compile with any Fortran compiler. Note that using C for the comment character or any symbol but the & for the continuation character will not be portable. A related issue with ensuring that code does not exceed the 71st column is that the CPP macro definitions in the Fortran include files may be longer than the name of the macro, thus pushing characters that appear to be with the 71st column past the 71st column. For example, depending on the Fortran compiler features and PETSc options, `PetscScalar` may be defined as `real(kind=selected_-real_kind(10))`, making user declarations such as

```
      PetscScalar   ainput,broot,ccase,dnile,erank
```

illegal with the fixed format.

Representing C objects in Fortran: PETSc is designed around a collection of abstract objects that have a variety of back-end implementations. For example, the `Mat` object in PETSc that represents linear operators is represented in the users C code as

```
typedef struct _p_Mat*          Mat;
```

This representation allows encapsulating the details of the matrix implementations outside the scope of the user code. The actual `_p_Mat` C struct contains a variety of data records as well as function pointers that implement all the matrix functionality for a particular matrix implementation. We provide two ways of mapping the `Mat` object to Fortran. In the traditional approach we use the fact that all Fortran variables are passed by pointer (i.e., the address of the variable is passed to the subroutine). On the Fortran side the objects are then just

```
#define Mat PetscFortranAddr
```

where, as before, `PetscFortranAddr` is either a 32- or 64-bit integer depending on the size of the memory addresses. A drawback to this approach is that in Fortran all PETSc objects are of the same type, so that the Fortran compiler cannot detect a type mismatch. For example, calling `MatMult()` with a vector object would not be flagged as incorrect. Hence we provide an alternative configure time approach where each PETSc object family is defined by a Fortran derived type and utilizes modules.

```
use petscmat
type(Mat) A
```

The corresponding definition in the PETSc module is simply

```
type Mat
  PetscFortranAddr:: v
end type Mat
```

Again the simplicity of the Fortran pass-by-pointer argument handling means that what is actually passed to a C stub is again an integer large enough to hold the PETSc object (which is, of course, a pointer). In fact, this definition allows the same Fortran application to refer to a `Mat` in some files using the traditional approach (as an integer) and in other files using the modern approach (as a Fortran derived type). With Fortran 2003 one no longer needs to use an appropriately sized integer to hold the C pointer in Fortran. Instead, one can use the construct

```
use iso_c_binding
type(c_ptr) :: A
```

to directly hold the C object pointer, or one can use

```
use iso_c_binding
type Mat
  type(c_ptr) :: v
end type Mat
```

Handling function callbacks in Fortran: PETSc users writing in C employ function callbacks to utilize much of the functionality of PETSc. For example, to use the nonlinear solvers, the user provides a C function that defines the nonlinear system,

```
PetscErrorCode func(SNES snes,Vec x,Vec f,void* ctx)
{
  /* evaluate a mathematical function putting the result into ctx
}
```

In their main program, after they have created a PETSc nonlinear solver object (called a SNES), they call

```
SNESSetFunction(snes,r,func,ctx);
```

The SNES object stores the function pointer and function context. Then whenever the PETSc nonlinear solver object needs to evaluate the nonlinear function, it simply calls the function pointer with appropriate arguments. Since the function pointer is stored in the SNES object, multiple solvers can work independently each with its own user functions. The user interface for Fortran is almost identical to that used from C. The user provides a Fortran function, for example,

```
subroutine func(snes,x,f,ctx,ierr)
SNES snes
Vec x,f,
type(fctx) ctx
PetscErrorCode ierr
! evaluate a mathematical function putting the result into ctx
return
end
```

and in the main program has

```
call SNESSetFunction(snes,r,func,ctx);
```

The PETSc code that supports this interface is essentially[1] the following.

```
static struct {
  PetscFortranCallbackId function;
  PetscFortranCallbackId destroy;
  PetscFortranCallbackId jacobian;
} _cb;
static PetscErrorCode oursnesfunction(SNES snes,Vec x,Vec f,
                                                     void *ctx)
{
  PetscObjectUseFortranCallback(snes,_cb.function,(SNES*,Vec*,
                                Vec*, void*,PetscErrorCode*),
                                (&snes,&x,&f,_ctx,&ierr)));
}
void STDCALL snessetfunction_(SNES *snes,Vec *r,
```

[1]The PGI Fortran compiler introduces an additional hidden pointer argument that we removed from this example to simplify the exposition.

```
                         void (STDCALL *func)(SNES*,Vec*,Vec*,
                         void*,PetscErrorCode*),void *ctx,
                         PetscErrorCode *ierr)
{
  *ierr = PetscObjectSetFortranCallback((PetscObject)*snes,
                   PETSC_FORTRAN_CALLBACK_CLASS,&_cb.function,
                   (PetscVoidFunction)func,ctx);
  if (!*ierr) *ierr = SNESSetFunction(*snes,*r,oursnesfunction,
                                                          NULL);
}
```

The routine `PetscObjectUseFortranCallback()` records the Fortran function pointer and context in the SNES object, which are then retrieved when the user's Fortran function is called.

In addition to introducing `type(c_ptr)` as discussed above for handling C objects (pointers) in Fortran, the 2003 standard introduced `type(c_funptr)` which represents a C function in Fortran. The standard also introduced Fortran function pointers and methods for converting between C and Fortran function pointers. Most important one can now directly declare a C function in the Fortran code and have the Fortran compiler automatically generate the stub code needed to call it from Fortran instead of requiring the user to manually provide the stub. This is done, for example, with

```
function MyCfunction(A) result(output) bind(C,name="cfunction")
  use iso_c_binding
  integer(c_int) :: A,output
end function
```

which declares a C function with a single C `int` input that returns an `int`. Fortran 2003 also provides `c_null_ptr` and `c_null_funptr` but still does not provide a generic `NULL` that has the flexibility to be passed anywhere a pointer is expected. The introduction of C bindings to the Fortran standard did not actually introduce any capability that was previously unavailable; it merely made the interface between C and Fortran simpler for those not familiar with the relatively straightforward issues outlined above.

Providing Fortran interface definitions: Since C provides function declarations that allow compile-time checking of argument types, Fortran 90 introduced interfaces that serve a similar purpose in Fortran.

9.3 Automatically Generated Fortran Capability

For a C package with only a handful of functions and data types, one can manually generate the appropriate Fortran function stubs (either using the traditional approach or with the C binding capability of Fortran 2003), Fortran interface definitions, and Fortran equivalent compile and runtime variables. However, for larger packages, especially those that evolve over time with more functionality, manually producing the capability and maintaining it is infeasible. Thus one must at least partially automate the process, much as SWIG [320] is used to generate Python stubs for C libraries. The PETSc team uses the Sowing package [322], developed by Bill Gropp for the MPICH package, to automatically generate much of the Fortran capability. Sowing generates both traditional Fortran stub functions and Fortran interface definitions for all C functions in the code that are "marked" by specific formatted comments. It does not handle character string arguments or function pointer arguments well, so those are handled in a partially manual manner. Since the traditional Fortran stub approach continues to work well for PETSc and is not a resource burden, we have not switched to using the Fortran 2003 C binding method for Fortran stubs. Thus we cannot say for sure that the Fortran 2003 C binding techniques would allow all the functionality from Fortran that PETSc's current techniques provide. Fully automatic generation of the C binding definitions, including proper support for `NULL` arguments and Fortran function pointers, would be a major exercise.

9.4 Conclusion

In PETSc, we have mapped all the C-based constructs needed by users, including enums, abstract objects, array pointers, null pointers, and function pointers (callbacks) to equivalent traditional Fortran and modern Fortran constructs, allowing Fortran PETSc users to utilize almost all the functionality of PETSc in their choice of Fortran standard. This support has substantially enlarged the user base for PETSc. We estimate that nearly one-third of our users work in Fortran, and we can provide them high quality numerical library support for modern algebraic solvers. As a result of automation of much of the process, the cost of PETSc Fortran support is significantly less than 10 percent of our development time. In addition, the Fortran support allows applications that are written partially in C and partially in Fortran, although we are not aware of many PETSc applications implemented in this way. Because of user and legacy demands, it is still important to support the full suite of F77, F90, F2003, F2011, and C interfaces. The advent of F2003 Fortran-C interoperability features, while a good addition, did not fundamentally change how PETSc supports Fortran users, nor did it allow us to discard outdated interfacing technology. Instead, it allowed us to enhance the Fortran support we already provided. The performance hit in using PETSc from Fortran rather than C for any nontrivial problems, consisting of only a small extra function call overhead, is negligible because of the granularity of the operations.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com