

*Communicating With the User in Numerical Routines*  
*Part II: An Iterative Linear Solver in F95 and F2x*

Aleksandar Donev

Physics and Astronomy Department, Michigan State University  
East Lansing, MI 48825  
*donev@pa.msu.edu*

## 1 *Introduction: Handle-Based Communication*

This paper is a continuation of “*Part I: OOP Approach in F2x*”, where I discussed the problem of communication between numerically-intensive routines and the user. In particular, I described in detail a replacement for the traditional reverse-communication mechanism which I called handle-based communication.

I got interested (read: troubled) by this problem because I was coding a conjugate-gradient iterative linear solver for a network optimization F95 library that I am writing. Iterative solvers are very important in numerical linear algebra, and several modern libraries implement them in F90/95 (such as PIM (Parallel Iterative Methods), or SMLIB (Sparse Matrix Library)), and in my experience none of these have devoted sufficient attention to the communication problem. Therefore, here I will give fully functional F95 code for a very simple iterative improvement linear solver, which emulates the OOP F2x approach to handle-based communication described in the first part of this paper.

## 2 *Iterative Linear Solvers: A Simple Example*

Iterative linear solvers are numerical methods for solving linear systems of the form,

$$Ax = b$$

where  $A$  is some matrix. A large class of iterative linear solvers require only one operation that explicitly involves the unknown matrix  $A$  — the evaluation of the *matrix-vector product*,

$$p = Ax$$

for some vector  $x$ . The aim of the iterative procedure is to continually improve a guess for the solution  $x$  so as to get the residual  $r = b - Ax$  to be small enough, as measured by some vector norm.

The simplest iterative method of all, which I will simply call **iterative improvement**, starts with some initial guess  $x_0$  and iteratively performs the update operation,

$$x_{n+1} \leftarrow x_n + r_n = x_n + (b - Ax_n)$$

until convergence is achieved. The need for the matrix-vector product is clear. I hope this explanation is sufficient to follow the code below.

## 2.1 The F95 source code

These are the code segments that will comprise our ready-to-compile F95 file, which can be obtained from the author via e-mail. FWEB creates this source code from the same file that created this document:

```
"WEAVE.f90" 2.1.1 ≡
```

```

< Precision 2.1.2 > // Precision constants
< Vector_Operations A.0.1 > // Level-1 BLAS-like operations
< Solver_Data_Types 3.1.6 > // Derived data-types definitions
< Iterative_Improvement 3.2.1 > // The iterative improvement method
< Diagonal_Linear_Systems 4.1.10 > // Diagonal linear systems
< TestProgram 4.2.1 > // Main program

```

```
< Precision 2.1.2 > ≡
```

```

MODULE Precision // A simple module that defines the working precision for real numbers
  INTEGER, PARAMETER, PUBLIC :: r_sp = KIND(0.0 · 100), r_dp = KIND(0.0 · 100D), r_wp = r_sp
  // Single, double and working precision
END MODULE Precision

```

This code is used in section 2.1.1.

## 3 *Implementation*: Iterative Linear Solvers

In this section I give code for an iterative improvement linear solver that uses a handle-based communication mechanism.

### 3.1 Derived Data Types

I will use derived data types extensively to simplify argument lists and to simply organize the code better. In fact, most of the design decisions will be made while declaring these data types. The first data-type, *Linear\_Solver*, declares the data structures needed by an iterative solver that are also of interest to the user as well (such as the total number of iterations, the desired tolerance, the final residual, or convergence status). It makes sense to package this data in a derived data type and make it all **PUBLIC**. In general, I will not concern myself too much with data encapsulation in this code — I think the user should have as much access and control as he might require in the most demanding situation imaginable (beginner users begone!).

The only important point to discuss about the *Linear\_Solver* data type concerns the format in which the vectors  $x$ ,  $b$ ,  $r$  and  $p$  are provided by and to the user. In usual implementations (as in P1M), these are assumed to be all simple vectors of size  $n$  and possibly declared as,

```
REAL (KIND = r_wp), DIMENSION (:), INTENT (INOUT) :: x, b, r, p
```

in the argument list of the iterative solver routine. There are several problems with this approach. The first is a simple lack of elegance in having to pass so many vectors in the argument list back and forth. But the serious problem with this implementation is the assumption that these are all one-dimensional vectors with the same number of elements, namely, the number of unknown variables  $n$ . In some applications, such as parallel codes where the matrix-vector product involves localized indirections, the matrix-vector product can be efficiently implemented only if additional, shadow, space is allocated to the arrays involved in the product, such as  $x$  and  $p$ . Of course, the iterative solver should only see  $n$  of these elements and update only these, while only the user will know that there is extra shadow space allocated.

The solution, at least formally, is easy. Make each of these arrays array pointers, so that the user can associate them with the proper (non-shadow) portion of the “real” (actual) arrays  $x$ ,  $b$ ,  $r$  and  $p$ . Pointers can also be packaged in derived data-types, so we avoid passing a lot of arguments as well and thus achieve more compact and clear code. But, most of us are well aware of the optimization dangers of array pointers, particularly because of aliasing. This is very important in iterative solvers which involve vector array statements such as  $x = x + r$ , which can not be optimized under aliasing assumptions. In the author’s view, the true solution to this is to add elements to F2x that will allow the user to tell the compiler that the pointers  $x$ ,  $b$ ,  $r$  and  $p$  can be treated as if they were autonomous arrays with their own separate storage space, as if for example they had the **ALLOCATABLE** and not **POINTER** attribute. Such ideas for restricting pointers have been discussed by the standard committee, but seemed not to have gotten enough attention. I urge committee members to reconsider the issue.

In this code, I solve the optimization problem with pointers by performing vector operations such as  $x = x + r$  inside dedicated routines that accept assumed-size arrays as arguments, placed here in the module *Vector\_Operations* (found at the end of this paper in a simplified version), so that the optimizer will compile these vector operations under a no-aliasing assumption. I considered making the arrays **ALLOCATABLE** instead of **POINTER** (to put these inside a derived data-type requires the ISO TR15581 allocatable components extension to F95, which this compiler supports), but found that my compiler currently treats these internally as **POINTER** (hasty implementations of TR’s begone!), so this did not help much.

So, without further to-do, here is the data-type for a generic iterative linear solver:

⟨Linear\_Solver 3.1.1⟩ ≡

```
INTEGER, PARAMETER, PUBLIC :: solver_free = 0, solver_inactive = -1, solver_error = -2,
    solver_calculating = 1, solver_communicating = 2 // Solver predefined statuses
INTEGER, PARAMETER, PUBLIC :: solution_free = 0, solution_error = -1, solution_unconverged = 1,
    solution_converged = 2 // Solution predefined statuses

TYPE Linear_Solver // Could be EXTENSIBLE in F2x
    INTEGER :: n_iterations = 0, max_iterations = 100 // Number of iterations
    INTEGER :: status = solver_free // Solver status
    INTEGER :: convergence = solution_free // Solution status
    REAL (KIND = r_wp) :: residual_norm = -1.0r_wp, residual_norm_tolerance = 1 · 10-5r_wp
    // Achieved and desired norm of r
    REAL (KIND = r_wp), DIMENSION (:), POINTER :: solution = _NULL, product = _NULL,
    residual = _NULL // Vectors x, p = Ax and r = b - p = b - Ax
ENDTYPE
```

This code is used in section 3.1.6.

To make a solvable linear system, represented here by a separate data type *Linear\_Solver*, we need to identify the right-hand-side vector  $b$  and the matrix  $A$ , and an object of type *Linear\_Solver*. The central problem of communication is the fact that the matrix  $A$  is not directly needed by the solver routines, but only the product  $p = Ax$  for some  $x$ 's, and in fact the solver routine will have no idea how  $A$  will be represented and stored.

The design decision made in my code is that the matrix-vector product will be supplied to the solver in the form of a procedure, with the interface given below in section 3.1.7 under **PROCEDURE** (*Multiply*). Packaging the matrix-vector product in a dummy argument procedure is hardly novel and is done in some newer libraries, such as PIM. However, as I explained in the earlier paper, the solver also needs to pass a handle for the matrix  $A$  to the multiplication routine. I emulate generic handles in F95 by using integers (Personal Identification Numbers–*PIN*) instead of F2x polymorphic pointers:

⟨Generic\_Matrix 3.1.3⟩ ≡

```
TYPE Generic_Matrix // In F2x: TYPE , EXTENSIBLE :: Generic_Matrix
    INTEGER :: PIN = 1 // In F95 I use plain default integers for handles
    CHARACTER (LEN = 32) :: name = "Matrix A" // For messages and such
    /* In F2x:
        PROCEDURE (Multiply), POINTER :: Multiplication
        would be included here for the user-supplied matrix-vector product (could be type-bound) */
ENDTYPE
```

This code is used in section 3.1.6.

⟨Linear\_System 3.1.4⟩ ≡

```
TYPE Linear_System // Could be EXTENSIBLE in F2x
```

```

    _TYPE(Generic_Matrix), POINTER :: matrix = _NULL    // A handle for the matrix A
    // In F2x: CLASS(Generic_Matrix), POINTER :: matrix = _NULL
    _TYPE(Linear_Solver), POINTER :: solver = _NULL
    // Could be CLASS(Linear_Solver) in F2x
    REAL(KIND = r_wp), DIMENSION(:), POINTER :: rhs = _NULL    // The rhs vector b
ENDTYPE

```

This code is used in section 3.1.6.

Finally, here is the collection of all data types associated with iterative linear solvers as defined above, packaged as usual into a module, called *Solver\_Data\_Types*, as well as the interface for the multiplication procedure `PROCEDURE(Multiply)`, whose presentation I had to delay until the end of this section because it depends on many user-defined types:

⟨*Solver\_Data\_Types* 3.1.6⟩ ≡

```

MODULE Solver_Data_Types
  USE Precision, ONLY:r_wp    // Single/double precision constants
  IMPLICIT NONE

  ⟨Generic_Matrix 3.1.3⟩    // A generic handle for matrices A
  ⟨Linear_Solver 3.1.1⟩    // A solver for  $Ax = b$ 
  ⟨Linear_System 3.1.4⟩    // A system  $Ax = b$  and its solver

END MODULE Solver_Data_Types

```

This code is used in section 2.1.1.

⟨*MultiplyInterface* 3.1.7⟩ ≡

```

INTERFACE    // In F2x: INTERFACE PROCEDURE()
  SUBROUTINE Multiply(system)    // Computes  $p = Ax$ 
    USE Solver_Data_Types    // In F2x: IMPORT :: Linear_System
    _TYPE(Linear_System), INTENT(INOUT) :: system    // Could be POINTER
    // Could be CLASS(Linear_System), INTENT(INOUT) :: system in F2x
  END SUBROUTINE Multiply
END INTERFACE

```

This code is used in section 3.2.3.

Please note that I choose to pass a whole linear system of type *Linear\_System* to the multiplication routine, which is somewhat excessive (in principle only the object of type *Linear\_Solver* and the handle for *A* need to be passed to the Multiplication routine). In fact, the `TYPE(Linear_System)` can be abolished and its three

or so components passed directly as arguments to the iterative solver routine. Here I effectively do this by packaging these arguments as pointers in the derived type *Linear\_System*. This simplifies experimentation with various combinations of solvers, matrix representations and in sensitivity analysis, and I use it here mostly because it simplifies the argument lists and allows easy modifications to the type *Linear\_System* without changing the interfaces.

Also, by making the type *Linear\_System* **EXTENSIBLE** and passing it as **CLASS** in the above interfacing, one can extend it, instead of extending the type *Generic\_Matrix*, with any data and procedures actually needed to perform the matrix-vector product. This approach is useful when the matrix *A* is not really represented explicitly and it doesn't make sense to pack it inside one data-type. In a library that I am writing, for example, the type *Linear\_System* is extended to a type *Network\_Dual\_System*, which represents a special type of system appearing in network optimization, and I ignore the type *Generic\_Matrix* altogether. By making both the *Generic\_Matrix* and *Linear\_System* extensible, maximal flexibility is gained at the cost of a modest increase in complexity.

### 3.2 Simple Iterative Improvement

Next I illustrate the usage of the above data-types for making a very simple iterative linear solver. Similar codes for real conjugate-gradient solvers can be obtained from the author, but they are not really hard to write once the principle is clear:

```

< IterativeImprovement 3.2.1 > ≡
  MODULE IterativeImprovement
    USE Precision, ONLY:r_wp
    USE Solver_Data_Types
    USE Vector_Operations
    IMPLICIT NONE
    PUBLIC :: IterativeImprovement    // The solver user-callable routine
  PRIVATE

  CONTAINS

    < IterativeImprovement 3.2.3 >

  END MODULE IterativeImprovement

```

This code is used in section 2.1.1.

The routine that actually performs the iterative improvement method is rather simple and I will not comment on it at length. The only part of the code which is still unsatisfactory in my opinion is the need to perform the vector operations, such as vector axpys and copies, in separate routines from the *Vector\_Operations* module. As I explained before, this is because of aliasing obstacles to optimization for array operations with array pointers:

⟨IterativeImprovement 3.2.3⟩ ≡

```

SUBROUTINE IterativeImprovement(system, Multiply)
  _TYPE (Linear_System), INTENT (INOUT) :: system    // Could be POINTER
  // In F2x could be CLASS (Linear_System)
  ⟨MultiplyInterface 3.1.7⟩    // In F95 we must pass the procedure explicitly

  _TYPE (Linear_Solver), POINTER :: solver
  solver ⇒ system % solver    // A shortcut

  /* One might want to perform some consistency checks here, such as proper association or
     allocation of the array pointers, and assign solver % status = solver_error in case of error. Here
     I skip this and just do the initialization: */
  solver % status = solver_calculating
  solver % convergence = solution_unconverged
  solver % n_iterations = 0

  /* Iterate  $x \leftarrow x + r = x + (b - p) = x + (b - Ax)$  until residual is small enough: */
Iterate: DO    // Start the iterative improvement process
  solver % n_iterations = solver % n_iterations + 1

  IF (solver % n_iterations > solver % max_iterations) THEN    // Too many iterations!
    solver % convergence = solution_unconverged
    EXIT Iterate
  END IF

  CALL Multiply(system)    // Let the user calculate  $p = Ax$ 
  // In F2x: CALL system % Multiplication(system). This should check for errors.
  IF (solver % status ≡ solver_error) THEN    // An error occurred in Multiply
    solver % convergence = solution_error
    EXIT Iterate
  END IF

  /* To avoid problems with aliasing analysis in this vector operation, I use the module
     VectorOperations and its routines that operate on assumed-size non-aliased arrays: */
  CALL VectorSubtraction(from = system % rhs, what = solver % product,
    difference = solver % residual)    //  $r = b - p = b - Ax$ 
  solver % residual_norm = SQRT(DOT_PRODUCT(solver % residual, solver % residual))
  //  $\|r\| = \sqrt{r^T r}$ 
  IF (solver % residual_norm < solver % residual_norm_tolerance) THEN    // Residual has converged
    solver % convergence = solution_converged
    EXIT Iterate
  END IF
  WRITE(*, *) "___Convergence: ", solver % n_iterations, solver % residual_norm

  CALL VectorAddition(first = solver % solution, second = solver % residual)    //  $x \leftarrow x + r$ 
END DO Iterate

  solver % status = solver_inactive    // Done!
  RETURN
END SUBROUTINE IterativeImprovement

```

This code is used in section 3.2.1.

## 4 *An Example: Diagonal Linear Systems*

Finally, to make the code complete, I give an example of using the iterative improvement iterative solver to solve a particularly simple kind of linear system — a diagonal system.

### 4.1 Diagonal Matrices

Consider the case when the matrix  $A$  is diagonal,  $A = \mathcal{D}(A)$ . In this case only one `REAL` vector for the diagonal is needed to represent the matrix:

⟨Diagonal\_Matrix 4.1.1⟩ ≡

```
TYPE Diagonal_Matrix // In F2x: TYPE , EXTENDS (Generic_Matrix) :: Diagonal_Matrix
REAL, DIMENSION (:), POINTER :: diagonal = _NULL // The diagonal  $\mathcal{D}(A)$ 
ENDTYPE
```

This code is used in section 4.1.10.

To make a workaround for the lack of generic pointers in F95, I resorted to using integers as handles. Now we need a method to convert (associate, relate) those integers (*PIN*) to a (pointer to a) diagonal matrix of type *Diagonal\_Matrix*. There are of course many ways to do this, from fancy search trees to linked lists, but here I adopt the simplest I know of — make an array of available objects of type *Diagonal\_Matrix* and then use the integer *PIN* as an index into this array to retrieve  $A$ . In a real-life problem one would probably require more dynamic handle management, such as, for example, making an array of available *pointers* to `TYPE (Diagonal_Matrix)`, but this is mostly an F95 workaround issue and is also problem-dependent. Here I choose to delegate handle-management to the main program (the user), mostly for simplicity:

⟨Diagonal\_Matrices\_Handles 4.1.3⟩ ≡

```
_TYPE (Diagonal_Matrix), DIMENSION (:), ALLOCATABLE, TARGET, PUBLIC :: diagonal_matrices
// The pool of available diagonal matrices
```

This code is used in section 4.1.10.

And here is the way to retrieve a pointer to an actual diagonal matrix from the handle in both the F95 workaround and F2x. In F2x I use the new feature of type selection, which allows one to branch according to

the run-time type of a polymorphic variable. The advantage of this approach is that the same multiplication procedure can be used with several different actual matrix types, which could be useful, for example, when testing different storage schemes for a given type of matrix:

⟨HandleToMatrix<sub>F95</sub> 4.1.5⟩ ≡

```
matrix ⇒ diagonal_matrices(system % matrix % PIN)
```

This code is used in section 4.1.8.

⟨HandleToMatrix<sub>F2x</sub> 4.1.6⟩ ≡

```
SELECT CASE_TYPE (system % matrix)
  CASE_TYPE IN(Diagonal_Matrix) // Point to the diagonal matrix of interest
  matrix ⇒ system % matrix
  CASE_TYPE DEFAULT // For now we only support diagonal matrices
  solver % status = solver_error
RETURN
ENDSELECT
```

Finally, here is the code for the diagonal matrix-vector multiplication routine that can be passed as an actual argument to *Iterative\_Improvement*. Again, I use the routines from *Vector\_Operations* to perform the actual calculation for efficiency reasons. This procedure should really be **PURE**, however, the F95 standard is (overly) restrictive in its limitations on **PURE** procedures for *possible* side-effects, so that the line, *matrix* ⇒ *diagonal\_matrices*(*system* % *matrix* % *PIN*) is not allowed by the standard, probably due to possible aliasing of the pointer *matrix* to the global data *diagonal\_matrices*. In fact, all pointer assignments using global data are forbidden in **PURE** procedures in the F95 standard. This rule, and other restrictions on **PURE** procedures, in my opinion, is overly restrictive. It should be clear that the routine below is free from side-effects, since the access to *diagonal\_matrices* is read-only, even though this may not be obvious to a compiler.

⟨DiagonalMultiply 4.1.8⟩ ≡

```
SUBROUTINE DiagonalMultiply(system) // "Almost" PURE
  _TYPE(Linear_System), INTENT(INOUT) :: system // Could be POINTER
  // Could be CLASS(Linear_System) in F2x

  _TYPE(Diagonal_Matrix), POINTER :: matrix // The matrix A
  _TYPE(Linear_Solver), POINTER :: solver // A shortcut

  solver ⇒ system % solver
  ⟨HandleToMatrixF95 4.1.5⟩ // See HandleToMatrixF2x

  CALL VectorMultiplication(product = solver % product, first = matrix % diagonal,
    second = solver % solution) //  $p = Ax = D(A)x$ 
```

```
END SUBROUTINE DiagonalMultiply
```

This code is used in section 4.1.10.

And now we wrap all this up into a module called *Diagonal\_Linear\_Systems*:

```
<Diagonal_Linear_Systems 4.1.10> ≡  
MODULE Diagonal_Linear_Systems  
  USE Precision, ONLY:r_wp  
  USE Vector_Operations  
  USE Solver_Data_Types  
  USE Iterative_Improvement  
  IMPLICIT NONE  
  PUBLIC :: Diagonal_Matrix    // Derived data types declarations  
  PUBLIC :: DiagonalMultiply   // The multiplication routine for  $p = Ax$   
  PRIVATE  
  
  <Diagonal_Matrix 4.1.1>    // Diagonal matrix type definition  
  <Diagonal_Matrices_Handles 4.1.3> // A workaround not needed in F2x  
  
CONTAINS  
  
  <DiagonalMultiply 4.1.8>  
  
END MODULE Diagonal_Linear_Systems
```

This code is used in section 2.1.1.

## 4.2 Test Program

Although all the routines and data-types above were designed to make usage and argument lists simple, our main program is a bit lengthy. This is mostly because I delegated all allocations and handle management to the main program:

```
<TestProgram 4.2.1> ≡  
PROGRAM Forward_Communication  
  USE Precision, ONLY:r_wp  
  USE Solver_Data_Types  
  USE Iterative_Improvement  
  USE Diagonal_Linear_Systems  
  IMPLICIT NONE  
  
  INTEGER :: variable, n_variables = 0, ID_number = 1 // The number of unknowns
```

```

_TYPE(Generic_Matrix), POINTER :: handle = _NULL // Only needed in F95
_TYPE(Diagonal_Matrix), POINTER :: matrix = _NULL
_TYPE(Linear_Solver), POINTER :: solver = _NULL
REAL, DIMENSION (:), POINTER :: rhs
_TYPE(Linear_System), POINTER :: system = _NULL

WRITE(*, *) "Enter the number of unknowns:"
READ(*, *) n_variables

/* Allocate some of the required resources: */
ALLOCATE(handle) // In F2x ALLOCATE(matrix)
ALLOCATE(solver)
ALLOCATE(system)
ALLOCATE(rhs(n_variables))

/* Handle management in F95 workaround: */
ALLOCATE(diagonal_matrices(1 : 10)) // Reserve handles for 10 diagonal matrices in F95
ID_number = 1 // Only one system for now
handle % PIN = ID_number // Save the PIN number
matrix => diagonal_matrices(ID_number) // Store the matrix to simplify typing

/* Create a specific solver: */
ALLOCATE(solver % solution(n_variables), solver % product(n_variables),
solver % residual(n_variables))
solver % max_iterations = 50
solver % residual_norm_tolerance = 1.0 · 10-3r_wp
CALL RANDOM_NUMBER(solver % solution) // Random initial guess for x

/* Allocate the remaining arrays and assign numerical values: */
ALLOCATE(matrix % diagonal(n_variables))
CALL RANDOM_NUMBER(matrix % diagonal) // Random diagonal  $\mathcal{D}(A)$ 
rhs = 1.0r_wp // Take  $b = 1$  in this test

/* Now we can assemble the system and solve it, and display the results: */
system % matrix => handle // In F2x: system % matrix => matrix
system % solver => solver
system % rhs => rhs
CALL IterativeImprovement(system = system, Multiply = DiagonalMultiply)
// In F2x: CALL IterativeImprovement(system)

IF (solver % status ≠ solver_error) THEN // Successful exit from IterativeImprovement
IF (solver % convergence ≡ solution_converged) THEN
WRITE(*, *) "The iterative method converged in ", solver % n_iterations, " iterations"
ELSE
WRITE(*, *) "The iterative method did not converge!"
END IF
WRITE(*, *) "Returned solution:", solver % solution
WRITE(*, *) "Correct solution:", rhs / (matrix % diagonal)
ELSE
WRITE(*, *) "An error occurred in the iterative solver!!!"
END IF

END PROGRAM Forward_Communication

```

This code is used in section 2.1.1.

## 5 *Conclusions: What Have We Learned From This?*

In this paper I gave a full example F95 code that illustrates an effective solution to the problem of communication between numerical routines and the user. The code was annotated with all changes needed to convert it into F2x, which was the motivating language for the development of the code. Although the F95 workaround lacked elegance in certain portions, and both codes require some performance tuning, I believe the approach is superior to the widely-used reverse communication mechanism. Several relevant points:

- The operations needed from the user should be packed into a procedure that will be passed to the numerical routine as an actual argument. If possible, try to formulate these as **PURE** procedures. This allows for multiple solvers to run concurrently without interfering with one another. Using procedures for this seems like a safe practice and usually leads to good code organization.
- Other than the data the user needs to operate on and storage to return the results in (such as the vectors  $x$  and  $y$ ), *always pass a handle* to the user-supplied routine that identifies the problem being worked on. In F2x, polymorphic pointers are a perfect generic handle.
- Use derived data types to package data. However, since array components have to be **POINTER** arrays in F95, be careful and watch for optimization problems related to aliasing. A good solution is to use pointers and derived data types at the top (interface) level, but place the numerically intensive codes in routines coded with more “traditional” argument lists.

Finally, I want to again point to a few “features” of F95 that I have found very troublesome in related work and that should be (have been?) fixed in F2x:

- The prohibition of **ALLOCATABLE** arrays as type components and procedure arguments. This was fixed in writing with ISO TR15581, but at least one vendor has yet to implement this properly.
- The lack of procedure pointers. Not being able to make handles for procedures or to include handles for procedures in derived types makes the above handle-based communication scheme ineffective in numerous occasions. I would urge vendors to quickly add procedure pointers to their compilers.
- The lack of language features to limit or prohibit certain types of aliasing, such as to guarantee that an array pointer can be treated as if it is associated with an array with autonomous storage. Such features are not planned for F2x either, further blocking the wider use of array pointers, which I consider one of the greatest creations of F90.
- The overly restrictive restrictions on **PURE** procedures. It is my belief that it is the user’s responsibility to insure that a given procedure is free of side-effects, with the compiler overriding the user’s claim only if an unsafe side-effect can be proven. Possible side-effects should be limited to warnings in actual compilers, rather than prohibited by the standard.
- The prohibition against passing internal procedures as actual procedure arguments. I urge the J3 committee to reconsider this issue.

## 6 Acknowledgements

I would sincerely like to thank the editor Michael Metcalf for his contribution to the final appearance of this paper and especially John Reid for his helpful comments on the ideas developed here and numerous suggestions for improvement. I would also like to thank everyone on the `comp-fortran-f90` mailing list who helped me whenever I needed clarifications and guidance.

## 7 References

Here are some related websites and articles, although this list is by no means extensive:

**J3 committee** <http://www.j3-fortran.org/>

**comp-fortran archive** <http://www.jiscmail.ac.uk/lists/comp-fortran-90.html>

**PIM library** <ftp://ftp.mat.ufrgs.br/pub/pim/>

**SMLIB library** <ftp://ftp.ntnu.no/pub/smlib/>

**My SSCNO library** <http://computation.pa.msu.edu/N0/F90/>

**F2x introduction** *Object Orientation and Fortran 2002*, Malcolm Cohen, ACM Fortran Forum: Part I — 16, 3, December 1997, 5-10, Part II — 18, 1, April 1999, 14-21.

## A Vector Operations

This is just an auxilliary module containing wrapper routines for certain vector operations that we needed in the above codes. One could for example make calls to level-1 BLAS routines here:

⟨ Vector\_Operations A.0.1 ⟩ ≡

```
MODULE Vector_Operations
  USE Precision, ONLY: r_wp
  IMPLICIT NONE
  PUBLIC :: VectorAddition, VectorSubtraction, VectorMultiplication // Just a few we need
  PRIVATE
CONTAINS
  PURE SUBROUTINE VectorAddition(first, second, sum)
    REAL (KIND = r_wp), DIMENSION (:), INTENT (INOUT) :: first
    REAL (KIND = r_wp), DIMENSION (:), INTENT (IN) :: second
    REAL (KIND = r_wp), DIMENSION (:), INTENT (OUT), OPTIONAL :: sum
    IF (PRESENT(sum)) THEN
      sum = first + second
    END IF
  END SUBROUTINE
```

```

ELSE
  first = first + second
END IF

END SUBROUTINE VectorAddition

PURE SUBROUTINE VectorSubtraction(from, what, difference) // No aliasing should be present
  REAL (KIND = r_wp), DIMENSION (:), INTENT (IN) :: from, what
  REAL (KIND = r_wp), DIMENSION (:), INTENT (OUT) :: difference

  difference = from - what

END SUBROUTINE VectorSubtraction

PURE SUBROUTINE VectorMultiplication(first, second, product)
  REAL (KIND = r_wp), DIMENSION (:), INTENT (IN) :: first, second
  REAL (KIND = r_wp), DIMENSION (:), INTENT (OUT) :: product

  product = first * second

END SUBROUTINE VectorMultiplication

END MODULE Vector_Operations

```

This code is used in section 2.1.1.