Fortran 95 Interface to MATLAB® API

Version 1.01

December 19, 2009

By James Tursa

1) Short Tutorial on Fortran Pointers

1.1) Contents of a Fortran Pointer

Conceptually, a Fortran pointer consists of the following:

1) The type of target being pointed to (class)
2) A base address of the target data being pointed to
3) The rank of the target (number of dimensions)
4) The dimension extents of the target (size of each dimension)
5) The stride of each dimension (e.g. every other element, etc.)
6) Automatically de-referenced in contexts that require an array

1.2) Comparison to C Pointers

Note how Fortran pointers are different from C pointers. C pointers basically have 1 and 2 from the above list and nothing else. You have to manually keep track of 3 – 5 off to the side in C code, and for 6 you have to explicitly dereference the C pointer when you want to access the target. A syntax comparison is as follows:

	Fortran	C
Declare 1D Pointers	real(8), pointer :: pA(:), pB(:)	double *pA, *pB;
Set pointer to null	pA => null()	pA = NULL;
Check for non-null	if(associated(pA)) then etc.	if(pA != NULL) { etc.
Declare 1D Arrays	real(8), target :: A(6), B(6)	double A[6], B[6];
Point to a 1D Array	pA => A	pA = A;
Point to part of 1D Array	pA => A(2:)	pA = &A[1];
Copy contents into target	pA = B(2:)	for(i=0; i<4; i++) { pA[i] = B[i+1]; }
Accomplish same thing with array section	A(2:) = B(2:)	for(i=0; i<4; i++) { A[i] = B[i+1]; }
Point to a 1D Array	pA => A	pA = A;
Copy pointer (not target)	pB => pA	pB = pA;
Size of target	size(pB)	(not available from pB)
Point to non-contiguous part (every other element)	pA => A(1:5:2)	(not directly available in C)

1.3) Fortran Pointer Assignment Operator =>

Note the special syntax for Fortran pointer assignment. The => operator points the left-hand-side to the right-hand-side (no data copy takes place), whereas the = operator copies the array contents of the right-hand-side into the target of the pointer on the left-hand-side (i.e., a data copy). In other words, using the = operator (or any operation that takes an array argument) automatically dereferences the pointer. Note that any object being pointed to must have the target attribute.

The rank (number of dimensions) of a Fortran pointer, like an array, is fixed at compile time and cannot be changed. So a 2D pointer will remain a 2D pointer throughout your code … you cannot use it to point to a 3D array (although you *can* point it to a 2D array slice of a 3D array).

2) MatlabAPI Extensions to MATLAB Supplied Routines

2.1) Philosophy of the Extensions

Each of the MatlabAPImex.f, MatlabAPImat.f, MatlabAPIeng.f, and MatlabAPImx.f files contains header information on the routines contained in the files. For example, MatlabAPImex.f contains interface information for all of the regular MATLAB API functions beginning with the letters mex. There are also additional related routines in the file. Typically these additional routines are Fortran versions of routines that The Mathworks supplied in the C interface but left out of the Fortran interface, but there are also a few extras for which there isn't a C counterpart. The same comments apply to the other three files. These routines typically begin with the characters mx, mex, mat, or eng just like the MATLAB API routines. Documentation for their use can be found in the source code files.

2.2) Extensions to mex routines

2.2.1) Stock MATLAB Fortran API Mex Routines

The interfaces for all of the MATLAB supplied Fortran mex routines are in the file MatlabAPImex.f in the module named MatlabAPImex. These interfaces give a very basic protection against misuse through number and type of arguments, but nothing else. Since all of the pointer types in the MATLAB Fortran API functions are represented with mwPointer (just an integer(4) or integer(8)), there is no guard against passing incorrect pointer types to the routines. E.g., passing the result of a mxGetPr call (really a real(8) pointer but represented as an integer(4)) to a routine expecting a mxArray pointer (also represented as an integer(4)) like mexMakeArrayPersistent will not be caught. Interfaces for the following MATLAB supplied Fortran routines are in the module MatlabAPImex:

mexCallMATLAB	mexIsLocked
mexCallMATLABWithTrap	mexLock
mexErrMsgIdAndTxt	mexMakeArrayPersistent
mexErrMsgTxt	mexMakeMemoryPersistent
mexEvalString	mexPrintf
mexEvalStringWithTrap	mexPutVariable
mexFunctionName	mexSetTrapFlag
mexGetVariable	mexUnlock
mexGetVariablePtr	mexWarnMsgIdAndTxt
mexIsGlobal	mexWarnMsgTxt

2.2.2) Replacement MATLAB Fortran API Mex Routines

The MATLAB supplied C API contains routines that are curiously missing from the Fortan API. To remedy this I have written replacement routines for them. The interfaces for the replacement routines were patterned after their C counterparts and the functionality is the same as their C counterparts. These replacement routines in the MatlabAPImex module, with source code, are as follows:

mexGet	mexSet

2.2.3) Additional MATLAB Fortran API Mex Routines

I added a few additional routines that fit in the mex group. They are:

mexCreateSparseLogicalMatrix (works like mxCreateSparseLogicalMatrix)
mexPrint (works like mexPrintf except adds a newline at the end)
fpMexGetNames (returns a list of all workspace variable names)

The first two routines above work just like the MATLAB doc for the similar functions listed. The third function listed above is brand new. Its signature is as follows:

```
function fpMexGetNames( ) result(fp)
character(len=63), pointer :: fp(:)
```

fpMexGetNames returns a pointer to character string array containing the names of all the variables in the workspace. The dynamic memory allocation for the character string array uses mxMalloc in the background, so to free the memory you have to call a special routine contained in the MatlabAPImx module. The typical use would be:

```
use MatlabAPImex
use MatlabAPImx
character(len=63), pointer :: fp(:)
fp => fpMexGetNames( ) ! be sure to use the => operator
! use the list
call fpDeallocate(fp)
```

If there are no variables in the workspace then fpMexGetNames returns a null pointer, which can be checked using the Fortran intrinsic function `associated`.

2.3) Extensions to mat routines

2.3.1) Stock MATLAB Fortran API Mat Routines

The interfaces for all of the MATLAB supplied Fortran mat routines are in the file MatlabAPImat.f in the module named MatlabAPImat. These interfaces give a very basic protection against misuse through number and type of arguments, but nothing else. Since all of the pointer types in the MATLAB Fortran API functions are represented with mwPointer (just an integer(4) or integer(8)), there is no guard against passing incorrect pointer types to the routines. Interfaces for the following MATLAB supplied Fortran routines are in the module MatlabAPImat:

matClose
matDeleteVariable
matGetDir
matGetNextVariable
matGetNextVariableInfo
matGetVariable
matGetVariableInfo
matOpen
matPutVariable
matPutVariableAsGlobal

2.3.2) Additional MATLAB Fortran API Mat Routines

I added a new routine that fits in the mat group intended as an easier-to-use replacement for matGetDir. It is:

fpMatGetNames (returns a list of all mat file variable names)

This function is brand new. Its signature is as follows:

```
function fpMatGetNames(mfp) result(fp)
character(len=63), pointer :: fp(:)
mwPointer, intent(in) :: mfp
```

fpMatGetNames returns a pointer to character string array containing the names of all the variables in the mat file. The dynamic memory allocation for the character string array uses mxMalloc in the background, so to free the memory you have to call a special routine contained in the MatlabAPImx module. The typical use would be:

```
use MatlabAPImat
use MatlabAPImx
character(len=63), pointer :: fp(:)
mwPointer mfp
mfp = matOpen("filename","mode") ! replace arguments with actuals
fp => fpMatGetNames(mfp) ! be sure to use the => operator
! use the list
call fpDeallocate(fp)
```

If there are no variables in the mat file then fpMatGetNames returns a null pointer, which can be checked using the Fortran intrinsic function `associated`.

2.4) Extensions to engine routines

2.4.1) Stock MATLAB Fortran API Engine Routines

The interfaces for all of the MATLAB supplied Fortran engine routines are in the file MatlabAPIeng.f in the module named MatlabAPIeng. These interfaces give a very basic protection against misuse through number and type of arguments, but nothing else. Since all of the pointer types in the MATLAB Fortran API functions are represented with mwPointer (just an integer(4) or integer(8)), there is no guard against passing incorrect pointer types to the routines. Interfaces for the following MATLAB supplied Fortran routines are in the module MatlabAPIeng:

engClose	engOpen
engEvalString	engOutputBuffer
engGetVariable	engPutVariable

2.4.2) Additional MATLAB Fortran API Eng Routines

I added a few additional routines that fit in the engine group. They are:

engCallMATLAB
engCreateLogicalSparseMatrix
engGet
engSet
fpEngGetNames

The engCallMATLAB, engCreateLogicalSparseMatrix, engGet, and engSet functions work the same as their mexCallMATLAB, mxCreateLogicalSparseMatrix ,mexGet, and mexSet counterparts except with the addition of the engine pointer argument, The fpEngGetNames function is brand new. Its signature is as follows:

```
function fpEngGetNames(ep) result(fp)
character(len=63), pointer :: fp(:)
mwPointer, intent(in) :: ep
```

fpEngGetNames returns a pointer to character string array containing the names of all the variables in the engine workspace. The dynamic memory allocation for the character string array uses mxMalloc in the background, so to free the memory you have to call a special routine contained in the MatlabAPImx module. The typical use would be:

```
use MatlabAPIeng
use MatlabAPImx
character(len=63), pointer :: fp(:)
mwPointer ep
ep = engOpen("")
! code to populate the engine workspace
fp => fpEngGetNames(ep) ! be sure to use the => operator
! use the list
call fpDeallocate(fp)
```

If there are no variables in the engine workspace then fpEngGetNames returns a null pointer, which can be checked using the Fortran intrinsic function `associated`.

2.5) Extensions to mx routines

2.5.1) Stock MATLAB Fortran API Mx Routines

The interfaces for all of the MATLAB supplied Fortran mx routines are in the file MatlabAPImx.f in the module named MatlabAPImx. These interfaces give a very basic protection against misuse through number and type of arguments, but nothing else. Since all of the pointer types in the MATLAB Fortran API functions are represented with mwPointer (just an integer(4) or integer(8)), there is no guard against passing incorrect pointer types to the routines. E.g., passing the result of a mxGetPr call (really a real(8) pointer but represented as an integer(4)) to a routine expecting a mxArray pointer (also represented as an integer(4)) will not be caught. Interfaces for the following MATLAB supplied Fortran routines are in the module MatlabAPImx:

mxAddField
mxCalcSingleSubscript
mxCalloc
mxClassIDFromClassName
mxCopyCharacterToPtr
mxCopyComplex16ToPtr
mxCopyComplex8ToPtr
mxCopyInteger1ToPtr
mxCopyInteger2ToPtr
mxCopyInteger4ToPtr

mxCopyPtrToCharacter
mxCopyPtrToComplex16
mxCopyPtrToComplex8
mxCopyPtrToInteger1
mxCopyPtrToInteger2
mxCopyPtrToInteger4
mxCopyPtrToPtrArray
mxCopyPtrToReal4
mxCopyPtrToReal8
mxCopyReal4ToPtr
mxCopyReal8ToPtr
mxCreateCellArray
mxCreateCellMatrix
mxCreateCharArray
mxCreateCharMatrixFromStrings
mxCreateDoubleMatrix
mxCreateDoubleScalar
mxCreateNumericArray
mxCreateNumericMatrix
mxCreateSparse
mxCreateString
mxCreateStructArray
mxCreateStructMatrix
mxDestroyArray
mxDuplicateArray
mxFree
mxGetCell
mxGetClassID
mxGetClassName
mxGetData
mxGetDimensions
mxGetElementSize
mxGetEps
mxGetField
mxGetFieldByNumber
mxGetFieldNameByNumber
mxGetFieldNumber
mxGetImagData
mxGetInf
mxGetIr
mxGetJc
mxGetM
mxGetN
mxGetNaN
mxGetNumberOfDimensions
mxGetNumberOfElements
mxGetNumberOfFields
mxGetNzmax
mxGetPi
mxGetPr
mxGetProperty
mxGetScalar
mxGetString
mxIsCell
mxIsChar
mxIsClass
mxIsComplex
mxIsDouble
mxIsEmpty
mxIsFinite
mxIsFromGlobalWS
mxIsInf
mxIsInt16
mxIsInt32
mxIsInt64
mxIsInt8
mxIsLogical
mxIsNaN
mxIsNumeric
mxIsSingle
mxIsSparse
mxIsStruct
mxIsUint16
mxIsUint32
mxIsUint64
mxIsUint8
mxMalloc
mxRealloc
mxRemoveField
mxSetCell
mxSetData
mxSetDimensions
mxSetField
mxSetFieldByNumber
mxSetImagData
mxSetIr
mxSetJc
mxSetM
mxSetN
mxSetNzmax
mxSetPi
mxSetPr

Even though interfaces exist for all of the above routines, many of them are rendered obsolete because of the Fortran pointer routines included later in the module.

2.5.2) Replacement MATLAB Fortran API Mx Routines

The MATLAB supplied C API contains logical routines that are curiously missing from the Fortan API. To remedy this I have written replacement routines for them. The interfaces for the replacement routines were patterned after their C counterparts and the functionality is the same as their C counterparts. Since Fortran allows for default logical, logical(1), logical(2), and logical(4) types, I have written routines for each of these types. These replacement routines in the MatlabAPImx module, with source code, are as follows:

mxCopyPtrToLogical
mxCopyPtrToLogical1
mxCopyPtrToLogical2
mxCopyPtrToLogical4
mxCopyLogicalToPtr
mxCopyLogical1ToPtr
mxCopyLogical2ToPtr
mxCopyLogical3ToPtr
mxCreateLogicalArray
mxCreateLogicalMatrix
mxCreateLogicalScalar
mxCreateLogical1Scalar
mxCreateLogical2Scalar
mxCreateLogical4Scalar
mxGetLogicalScalar
mxGetLogical1Scalar
mxGetLogical2Scalar
mxGetLogical4Scalar
mxGetLogicals
mxIsLogicalScalar
mxIsLogicalScalarTrue

Some Fortran compilers also support types real(16), complex(16), and integer(8). Even though MATLAB does not support these types, it is possible that a 3rd party has written a class for them. Thus I have written some copy routines to support them. For Fortran compilers that do not support these types, use the –DSMALLMODEL flag on the mex command line when compiling to delete them from the module. The routines are:

mxCopyComplex32ToPtr
mxCopyInteger8ToPtr
mxCopyPtrToComplex32
mxCopyPtrToInteger8
mxCopyPtrToReal16
mxCopyReal16ToPtr

Finally, there are some character routines to support certain functions for that data type. They are:

mxCopyCharsToCharacter
mxCopyCharacterToChars
mxGetChars

For the character routines, it is important to understand their intended use. The MATLAB supplied routines mxCopyPtrToCharacter and mxCopyCharacterToPtr listed in section 2.5.1 are intended to be used with pointers to C-style strings (null terminated strings), whereas the mxCopyCharsToCharacter and mxCopyCharacterToChars routines are intended to be used with normal Fortran character strings (not null terminated). The latter two work with the mxGetChars function, which gets the pointer to the data area of a char mxArray variable. The former two work with the matGetDir and mxCopyPtrToPtrArray routines, and are rendered obsolete by the fpMatGetNames function.

2.5.3) Additional MATLAB Fortran API Mx Routines Motivation

As mentioned previously, the interfaces supplied for the stock MATLAB API routines and additional routines listed earlier in this document do not give anything more than a basic protection against misuse. Also, to use them in a memory efficient manner it is necessary to employ the %VAL( ) construct, which requires using implicit interfaces to routines to get conforming array size behavior. For example, a typical use in a mex routine for getting a 2D array and passing it down to a subroutine for use would be:

```
#include "fintrf.h"
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
!-ARG
      mwPointer plhs(*), prhs(*)
      integer*4 nlhs, nrhs
!-FUN
      mwPointer, external :: mxGetPr
      mwSize, external :: mxGetM, mxGetN, mxGetNumberOfDimensions
      integer*4, external :: mxIsDouble
!-LOC
      mwPointer pr
      mwSize M, N
!-----
      if( nrhs < 1 ) then
          call mexErrMsgTxt("Need one double 2D input")
      endif
      if( mxIsDouble(prhs(1)) == 0 .or.
     &    mxGetNumberOfDimensions(prhs(1)) /= 2 ) then
          call mexErrMsgTxt("Need one double 2D input")
      endif
      pr = mxGetPr( prhs(1) )
      M = mxGetM( prhs(1) )
      N = mxGetN( prhs(1) )
      call sub( %VAL(pr), M, N )
          !
      return
      end subroutine mexFunction
!-----
      subroutine sub( A, M, N )  ! implicit interface, passed shape
      mwSize M, N
      real(8) A(M,N)
          !
      return
      end subroutine sub
```

The above code gets the pointer to the real(8) data with the mxGetPr call and stores the value in the pr variable. Then a call to sub is made with %VAL(pr) to get the conforming array behavior in sub. A conforming array cannot be used in the calling routine mexFunction. While this works, it is a bit cumbersome to use. Also, if one wants to copy the prhs(1) data into Fortran arrays one is forced to write custom code patterned after the above code or use the routine mxCopyPtrToReal8. And if one wants to create an mxArray (e.g. plhs(1)) from a Fortran array one must again create custom code or use the mxCopyReal8ToPtr routine. There are many chances for a programmer to either pass the wrong pointer, or get the sizes incorrect, etc. in these calls. Any of these errors would typically result in memory corruption and a program bomb. To remedy this, I have written routines that use assumed shape Fortran pointers to directly get at

the data areas of an mxArray variable. Use of these routines makes the use of many of the stock MATLAB API routines obsolete. For example, the code listed above could be written as follows:

```
#include "fintrf.h"
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      use MatlabAPImex
      use MatlabAPImx
!-ARG
      mwPointer plhs(*), prhs(*)
      integer*4 nlhs, nrhs
!-LOC
      real(8), pointer :: A(:,:)
!-----
      if( nrhs < 1 ) then
          call mexErrMsgTxt("Need one double 2D input")
      endif
      A => fpGetPr( prhs(1) )
      if( .not.associated(A) ) then
          call mexErrMsgTxt("Need one double 2D input")
      endif
      call sub( A )
          !
      return
      contains
!-----
      subroutine sub( A )  ! explicit interface, assumed shape
      real(8) :: A(:,:)
          !
      return
      end subroutine sub

      end subroutine mexFunction
```

The fpGetPr call gets an assumed shape Fortran pointer that points directly at the pr data area of the mxArray prhs(1). The beauty of the above scheme is that the Fortran pointer A can be used just like it was an ordinary Fortran array. It can be used anywhere an array of the same type and rank can be used, and such use will automatically be de-referenced. That is, it would have the same effect as using the target directly. You can treat it like a Fortran array in the mexFunction routine, and you can pass it in an explicit interface to routines taking assumed shape array inputs. All of the type and size checking, as well as the messy use of the %VAL( ) construct, are buried in the fpGetPr routine. The user, in one line, essentially has a Fortran array to work with. The MatlabAPImx module in the MatlabAPImx.f file is special in that it contains dozens of new routines like fpGetPr that interface Fortran pointers with mxArray variables. Extensive use is made of generic function names whenever possible. There is no MATLAB API equivalent (Fortran or C) to these routines. The vast majority of these routines begin with the letters fp, indicating that they either return a Fortran pointer or take a Fortran pointer as an argument (or both). In the descriptions that follow, it is assumed that:

fp= a real(8) (or complex(8) per context) Fortran Pointer variable of the appropriate rank
mx = a MATLAB mxArray variable double class.

2.5.4) fpGetPr and fpGetPi, Pointing Directly at mxArray pr and pi data

```
fp => fpGetPr1(mx)        ! Returns 1D pointer to the real data area of mx
fp => fpGetPr2(mx)        ! Returns 2D pointer to the real data area of mx
        :
fp => fpGetPr7(mx)        ! Returns 7D pointer to the real data area of mx

fp => fpGetPi1(mx)        ! Returns 1D pointer to the imag data area of mx
fp => fpGetPi2(mx)        ! Returns 2D pointer to the imag data area of mx
        :
fp => fpGetPi7(mx)        ! Returns 7D pointer to the imag data area of mx
```

The number in the name indicates the rank (number of dimensions) of the returned pointer. A typical interface for one of the above routines is:

```
function fpGetPr1( mx ) result(fp)
real(8), pointer :: fp(:)
mwPointer, intent(in) :: mx
```

The above routines return pointers that point directly at the data areas of the mx variable *with the shape of the mx variable*! If you use fp in an assignment that changes fp (e.g., fp = 3.d0), then you will be changing the data area of mx directly. This works because both Fortran and MATLAB store multi-dimensional arrays in column order. Since there is no memory allocated by these routines (they point directly at the mx data areas), you must *not* try to deallocate fp when you are done.

fp and mx must be compatible in rank. That is, the rank of fp must be at least the number of dimensions of mx. The only exception to this is that the fpGetPr1 function can be used on any mx variable regardless of the number of dimensions. In that case fp will be pointing to the data area as if it were one long 1D array.

If anything doesn't match up properly (rank mismatch, class not double, etc.) then a null pointer will be returned (i.e, the pointer returned will not be associated). E.g., this can be checked as follows:

```
fp => fpGetPr1(mx)
if( .not.associated(fp) ) then
    ! take appropriate action
endif
```

It is recommended that you always check the return value of any fp function to make sure the result is associated with a target.

2.5.5) fpGetPrCopy and fpGetPiCopy, Pointing at Copy of mxArray pr and pi data

```
fp => fpGetPrCopy1(mx)        ! Returns 1D pointer to copy of mx real data
fp => fpGetPrCopy2(mx)        ! Returns 2D pointer to copy of mx real data
        :
fp => fpGetPrCopy7(mx)        ! Returns 7D pointer to copy of mx real data
```

```
fp => fpGetPiCopy1(mx)          ! Returns 1D pointer to copy of mx imag data
fp => fpGetPiCopy2(mx)          ! Returns 2D pointer to copy of mx imag data
        :
fp => fpGetPiCopy7(mx)          ! Returns 7D pointer to copy of mx imag data

fp => fpGetPzCopy1(mx)          ! Returns 1D pointer to copy of real & imag data
fp => fpGetPzCopy2(mx)          ! Returns 2D pointer to copy of real & imag data
        :
fp => fpGetPzCopy7(mx)          ! Returns 7D pointer to copy of real & imag data
```

The above "copy" routines point to *copies* of the data areas of the mx variable. If you use fp in an assignment that changes the target of fp (e.g., fp = 3.d0), then you will *not* be changing the data area of mx, you will be changing the data of the copy instead. Since the memory for the copy is dynamically allocated, you must use the following to deallocate it when you are done:

```
call fpDeallocate(fp)
```

The fpGetPzCopy# functions return a complex pointer. Note that there are no fpGetPz# functions, only fpGetPzCopy# functions. That is because Fortran complex variables store the real and imaginary data interleaved, whereas MATLAB mxArray variables store the real and imaginary data in two separate pieces. Because of this, if is physically impossible to have a complex Fortran pointer pointing directly at mxArray variable data memory. A typical interface for one of the above functions is:

```
function fpGetPrCopy2( mx ) result(fp)
real(8), pointer :: fp(:,:)
mwPointer, intent(in) :: mx
```

2.5.6) fpGetDimensions, Pointing Directly at mxArray dimensions data

To get the dimensions of an mxArray variable directly you can use the following:

```
fp => fpGetDimensions(mx)
```

The result will be a pointer to a 1D array containing the dimension data. Since the result is pointing directly at the mx dimension data (it is not an allocated copy), it should be regarded as read-only and you should not try to deallocate it. The interface for this function is:

```
function fpGetDimensions( mx ) result(fp)
mwSize, pointer :: fp(:)
mwPointer, intent(in) :: mx
```

2.5.7) fpGetCells#, Pointing Directly at mxArray cell data

To get at the mxArray pointers of a cell array you can use the following:

```
fp => mxGetCells1(mx)
```

This would be for a 1D mx (or to treat an nD mx variable as a 1D variable). There are other obvious functions mxGetCells# for arrays of different dimensionality. No need to fuss around with all that complicated indexing stuff in the regular MATLAB API, just use fp as an array of mxArray pointers. Since the result is pointing directly at the mx data (it is not an allocated copy), it should be regarded as read-only and you should not try to deallocate it. A typical interface for one of the functions is:

```
function fpGetCells1( mx ) result(fp)
mwPointer, pointer :: fp(:)
mwPointer, intent(in) :: mx
```

2.5.8) mxArray, Creating an mxArray Variable from Copy of Fortran Variable

The above paragraphs are concerned with getting data out of mxArrays into Fortran variables. What about the other way? For that, there is a single generic function called, appropriately, mxArray. It takes a Fortran pointer or array as input and constructs an mxArray variable out of it and returns the pointer to that mxArray. Its use is very straightforward:

```
mx = mxArray(X [,Y] [,orient='row'] )
```

X can be a real(8) or complex(8) pointer or variable. The dimensionality of the resulting mxArray will be taken directly from X. If X is real(8) and Y is also given, then Y must be real(8) and Y will be taken to be the imaginary part and a complex mxArray will be constructed. In that case X and Y must agree exactly in shape (same number of dimensions and same dimension extents). If X is a 1D variable, then the result will be a column vector mxArray unless the optional orient='row' argument is present, in which case the result will be a row vector mxArray. The data areas of the resulting mxArray variable will be *copies* of the X and Y data. So if this is a temporary mxArray that is not returned to the MATLAB workspace via the plhs(*) variables, you should destroy it when you are done with it to free up the dynamically allocated memory:

```
call mxDestroyArray(mx)
```

Since mxArray is a generic function, there are many specific functions behind this generic name. These specific functions are not intended to be called directly by the user. However, to give a complete picture of the calling sequence, a typical interface for one of these function is as follows:

```
mwPointer function mxArray1double(A, B, orient) result(mx)
real(8), intent(in) :: A(:)
real(8), optional, intent(in) :: B(:)
character(len=*), optional, intent(in) :: orient
```

2.5.9) mxArrayHeader, Creating an mxArray Variable Directly from Fortran Variable

There is another generic function called mxArrayHeader. Its signature is similar to mxArray:

mx = mxArrayHeader(X [,Y] [,orient='row'])

mxArrayHeader works much the same as mxArray except that the data areas are *not copies* of X (and Y if present). Instead, the data areas point directly at the X (and Y if present) input and thus it is faster and more memory efficient than the mxArray function. However, this is also *very dangerous* and should only be used by experienced programmers who know what they are doing. Since X and Y are directly attached to the data areas of the mxArray variable created, you need to be very careful how you use this mxArray. If X and Y are regular Fortran memory, then the mxArray must not under any circumstances be returned to MATLAB via a plhs(*) argument. That would be mixing Fortran memory and MATLAB memory in an unstable fashion and will likely result in memory corruption and program bombing. Instead, the mxArray must be destroyed with the following routine before the mex routine returns control back to MATLAB:

```
call mxDestroyArrayHeader(mx)
```

This special mxDestroyArrayHeader routine first detaches the data pointers from the mx variable and then calls the regular mxDestroyArray routine on that.

So what is the purpose of the mxArrayHeader routine? Mainly it is useful for creating temporary mxArray variables for use with mexCallMATLAB or engCallMATLAB or matPutVariable where you know the call you make will not result in shared data. E.g., instead of doing this:

```
rhs(1) = mxArray(X)           ! A copy of X is made
k = mexCallMATLAB(1, lhs, 1, rhs, "sin")
Y => fpGetPrCopy(lhs(1))      ! A copy of the lhs(1) data is made
call mxDestroyArray(rhs(1))   ! Destroys header and data of rhs(1)
call mxDestroyArray(lhs(1))   ! Destroys header and data of lhs(1)
```

You can do this and eliminate some wasteful copying:

```
rhs(1) = mxArrayHeader(X)     ! Only header is created, no data copy
k = mexCallMATLAB(1, lhs, 1, rhs, "sin")
Y => fpGetPr(lhs(1))          ! Only pointer is returned, no data copy
call mxDestroyArrayHeader(lhs(1)) ! Only header is destroyed, not data
```

2.5.10) fpAllocate and fpAllocateZ, Allocating Variables with MATLAB memory manager

Fortran has a mechanism for dynamic memory allocation via the allocate and deallocate functions. Unfortunately, the MATLAB memory manager knows nothing about this memory so it cannot be garbage collected in the event of an abnormal program exit. Also, this memory cannot be attached directly to an mxArray variable. To remedy this situation, there are two memory allocation functions provided in the MatlabAPImx module that use the MATLAB API function mxMalloc in the background. They are:

fp => fpAllocate(n1 [,n2 [,n3 [,n4 [,n5 [,n6 [,n7]]]]]])

and

fp => fpAllocateZ(n1 [,n2 [,n3 [,n4 [,n5 [,n6 [,n7]]]]]])

The fpAllocate function returns a real(8) pointer of the appropriate rank, and the fpAllocateZ function returns a complex(8) pointer. When done with the memory both of these pointers can be deallocated as follows:

```
call fpDeallocate(fp)
```

Or, if desired, the memory can safely be attached to an mxArray variable data area since the memory is known to the MATLAB memory manager (remember, it originally came from an mxMalloc call). E.g.,

```
mx = mxArrayHeader(fp)
```

In this special case, since the memory behind the fp variable was created from the MATLAB mxMalloc function, the variable mx would be safe to use in any context … e.g. it would be safe to return it to the MATLAB workspace via the plhs(*) array.

2.5.11) fpReshape, Reshaping a Fortran Array or Pointer Without Copying

There is also a generic reshape function available called fpReshape. The Fortran language has, in fact, an intrinsic reshape function. Unfortunately, this intrinsic function always produces a copy of the input. When working with very large arrays this might not be desirable and in fact its use might cause a stack overflow and bomb your program. The fpReshape function does not produce a copy, it returns a Fortran pointer that points directly at the input so it is very fast and will not overflow your stack memory when working with very large arrays. The syntax for use is very straightforward:

fp => fpReshape(X, n1 [,n2 [,n3 [,n4 [,n5 [,n6 [,n7]]]]]])

Where X can be any rank and shape. X can be an array or a pointer to an array. The variable fp must be a real(8) or complex(8) pointer of appropriate rank. That is, if only n1 is supplied in the argument list, then fp must be a 1D pointer. If only n1 and n2 are supplied, then fp must be a 2D pointer. If X is real(8), then fp must be real(8). If X is complex(8), then fp must be complex(8). Etc. Just keep in mind that fp points to the memory associated with X, so changing fp will result in changing X. Since no new memory is allocate for fp, you must *not* try to deallocate it when you are done with fp. Note that the fpReshape routine works with Fortran arrays and pointers, not mxArrays. Even so, it is included in this package because of its relevance.

2.5.12) fpReal and fpImag, Pointing Directly at Real & Imag part of Complex Array

Another set of routines are the fpReal and fpImag functions. Their signatures are:

fp => fpReal(z)
fp => fpImag(z)

For a complex(8) input z, these functions return real(8) pointers to the real and imaginary parts of z. Note that these point directly at the z data, they are not copies. Thus you should *never* try to deallocate these fp pointer variables. To do so would likely result in memory corruption and your program bombing. The variable fp must be the same rank as z.

2.5.13) Troubleshooting

Using Fortran pointers for the first time can be a bit confusing. The pointer assignment operator => vs the regular assignment operator = takes some getting used to. It is with near 100% certainty that at some point you will make the following mistake (for the example assume mx is some pre-existing mxArray variable, e.g. prhs(1)):

```
real(8), pointer :: fp(:,:)
fp = fpGetPr2(mx)
```

This will result in a program bomb. Why? Because the regular assignment operator = was used instead of the pointer assignment operator =>. What the above statement attempts to do is copy the contents of the data area of mx into the target of fp. But since fp is garbage at that point in the program (it is not pointing to any valid memory yet) the program will bomb. In C this would be akin to copying data to a de-referenced C pointer before you had initialized the pointer to point to something valid (e.g., via a malloc call). The remedy for the above situation is to use the => pointer assignment operator:

```
real(8), pointer :: fp(:,:)
fp => fpGetPr2(mx)            ! used => instead of =
```

Also, you need to be cognizant of which routines allocate memory (and need deallocation when done) and which routines do not. If you attempt to deallocate a pointer that was not the result of dynamic memory allocation, the results are unpredictable and could result in a program bomb. E.g.,

These routines return pointers directed at the input. Do not deallocate the pointer they return:

fpGetPr#
fpGetPi#
fpGetCells#
fpGetDimensions
fpReshape
fpReal
fpImag

These routines return pointers directed at newly allocated memory. You should deallocate the pointer they return (use fpDeallocate) when done with it:

fpGetPrCopy#
fpGetPiCopy#
fpGetPzCopy#
fpAllocate
fpAllocateZ

Also, keep in mind that when destroying newly created mxArray variables, the mxArray function is typically paired with the mxDestroyArray subroutine, and the mxArrayHeader function is typically paired with the mxDestroyArrayHeader subroutine.

When compiling, it is possible that your compiler supports the %LOC( ) construct but not the LOC( ) function. If that is the case, compile with the –DPERCENTLOC option. E.g., for the MatlabAPImx module you would do this:

```
mex -c -DPERCENTLOC MatlabAPImx.f
```

2.5.14) Testing

The only currently tested configurations for the MatlabAPI code are PC 32-bit Windows XP with MATLAB versions R2006b - R2009b and the Compaq 6.1, Intel 9.1, and Intel 10.0 compilers. All other configurations are untested. The author would like to solicit help from the community in getting the MatlabAPI (and the self-building code in the build routines) to work under other configurations.

2.5.15) Updates Planned

- Bug fixes, of course.
- More extensive test routines.
- Expanded documentation and examples.
- Support for 64-bit and linux and mac machines, particularly for self building. But this will depend on other users willing to supply this code to me, since I do not have access to these machines for development or testing.

2.5.15) Updates Under Consideration

Future upgrades that I *may* consider depending on my time availability and the popularity of the Fortran 95 API:

- More Fortran pointer routines if I can think of any that make sense.
- Code for other classes (single, integer, logical, etc).
- Support for older versions of MATLAB, particularly versions prior to 7. My guess is that some of the MATLAB code and/or API functions I used might not be compatible. Again, this will depend on other users willing to help modify the code, since I do not have access to these older versions of MATLAB for development or testing.

2.5.15) Contact the Author

Feel free to post items of general interest to other users (bug reports, performance data, questions about usage or optimizations, etc) directly on the FEX of course. But if you have modified the code for your version of MATLAB (older version, non-PC machine, non-supported Fortran compiler, etc.) please feel free to contact me directly and I will try to incorporate them into future Fortran 95 API upgrades. You can reach me at

a#lassyguy%ho$mail_com (replace # with k, % with @, $ with t, _ with .)

James Tursa