Derived Types

Abstract

In this report we will explore how define and use derivate types in Fortran. We will implement a double complex matrix DERIVATED TYPE, define the function TRACE and ADJOINT and their INTERFACES and write a subroutine which will write the TYPE on file.

Theory

Code Development

The first thing to do is create a MODULE, called *matrices*, which will contain the definition of TYPE *doublecomplex_matrix*.

```
module matrices
implicit none
type doublecomplex_matrix
!defining the type: doublecomplex matrix
integer*2 :: nr
integer*2 :: nc
double complex, dimension (: , :), allocatable :: elements
double complex :: m_trace
double complex :: m_det
end type doublecomplex_matrix
```

The second step is create a subroutine *initializ_matrices* which inizialites a type doublecomplex_matrix, giving as imput an object of that type and the number of row and the number of column. It was necessary to define again the imput inside the subroutine. The subroutine creates a matrix such that at the entry (i,j) there is the d_complex number (i,j). This is realised using the counting variables which go through the matrix yy and xx: $AA\%elements(yy, xx) = complex\,(1d0 * yy, 1d0 * xx)$. It gives a default value 0 to the determinant and to the trace.
Given the number of rows and columns, it is necessary to allocate the matrix.
The command $contains$ is at the top of the list of subroutines and functions used in the module.

```
contains

subroutine initializ_matrices (AA, numrow, numcol)
!this subroutine initializes a type doublecomplex_matrix,
!given as imput. Also the numbers of rows and columns
!are given as imput. The subroutine creates a matrix such that
!at the entry (i,j) there is the d_complex number (i,j).
!It gives a default value 0 to the determinant and to the trace
integer yy, xx
type(doublecomplex_matrix) :: AA
integer*2 numrow, numcol
AA%nr= numrow
AA%nc= numcol
allocate (AA%elements(AA%nr, AA%nc))
do yy= 1, AA%nr
        do xx= 1, AA%nc
                AA%elements(yy,xx)=complex(1d0*yy,1d0*xx)
```

```
        end do
end do
AA%m_trace=(0d0,0d0)
AA%m_det=(0d0,0d0)
end subroutine initializ_matrices
```

Another subroutine, called *print_matrices*, prints a type doublecomplex_matrix, given as imput, on a file, which name is given as imput. What is printed is the following: the numbers of row and columns, the elements (in the proper order, via a do cycle), the trace and the determinant.

```
subroutine print_matrices (AA, namefile)
!This subroutine prints a type doublecomplex_matrix:
! the numbers of row and columns, the elements
!(in the proper order), the trace and the determinant.
!This is all printed on a file, which name is given as imput.
type(doublecomplex_matrix) :: AA
character (:), allocatable :: namefile
integer :: ii,jj

open(unit = 40, file = namefile, status = "unknown")
write (40,*) "NUMBER␣OF␣ROWS:", AA%nr
write (40,*) "NUMBER␣OF␣COLUMNS:", AA%nc
write (40,*) "ELEMENTS:"

!Following do cycle in order prints in the proper order
do ii=1, AA%nr
        write (40,*) (AA%elements(ii, jj), jj = 1, AA%nc)
end do

write (40,*) "TRACE:", AA%m_trace
write (40,*) "DET:", AA%m_det
close(40)
end subroutine print_matrices
```

Then we define a function, called *MatAdjoint*. This function, given a type doublecomplex_matrix in imput, gives as result a type doublecomplex_matrix which is the adjoint of the imput. The number of rows and columns are exchanged in the output matrix, which is the adjoint of the imput matrix.
In Fortran, functions seem to be are more rigid than the subroutines. It is necessary to dedicate the first parte of the function's code to defining the variables, because it is not possible to define variables later in the code. Moreover, it is necessary for the imputs to declare their intent as input, using the command $intent(in)$. After having specified its number of row and columns, it is also compulsory to allocate the output type doublecomplex_matrix.

```
function MatAdjoint (CC) result (BB) !this function,
```

```
!given a type doublecomplex_matrix in imput, gives as
!result a type doublecomplex_matrix which is the adjoint
!of the imput. (number of rows and columns are exchanged
!and the output matrix is the adjoint of the imput matrix)
integer :: kk, qq
type(doublecomplex_matrix), intent(in) :: CC
type(doublecomplex_matrix)  :: BB
BB%nr=CC%nc
BB%nc=CC%nr

allocate (BB%elements(BB%nr, BB%nc)) !in a function
!it is necessary to allocate the output, if it needs it.

do kk= 1, CC%nr
        do qq= 1, CC%nc
                BB%elements(qq,kk)=conjg(CC%elements(kk,qq))
        end do
end do

end function MatAdjoint
```

Then we define a function, called *MatTrace*. This function neees a type doublecomplex_matrix as a imput. If it is a square matrix, the trace is as usual the sum of the elements along the major diagunal. If it is rectangular, a warning message is printed on the terminal and a zero value is returned. This is done via an *if*, which checks if the difference of the numbers of columns and rows (in absolute value) is less than 0.5 or not. If that difference is less than 0.5 the trace is computed. It was chosen to do not use a statement like "if the number of column is egual to the number of rows then compute the trace" in order to avoid possible errors due to the computer's imprecision. Moreover, the boundary is not 1 for the same reason.

```
function MatTrace (DD) result (trace)
!This function, given a type doublecomplex_matrix
!as a imput, gives as result the trace of that matrix.
!If it is a square matrix, the trace is computed.
!If it is rectangular, a warning is printed and
!a zero value is returned.

integer :: tt
double complex trace
type(doublecomplex_matrix), intent(in) :: DD
trace = (0d0, 0d0)

!checking if the matrix is squared
if (abs(DD%nr-DD%nc) < 0.5) then
        do tt= 1, DD%nr
                trace= trace + DD%elements(tt,tt)
```

```
            end do
        end if
        end function MatTrace
```

Those functions are provided with an interface, which is written before them, just after the definition of the type doublecomplex_matrix. $.Adj.$ stands for MatAdjoint function. $.Tr.$ stands for MatTrace function.

```
        interface operator (.Adj.)
        !.Adj. stands for ADJOINT function
        module procedure MatAdjoint
        end interface

        interface operator (.Tr.)
        !.Tr. stands for ADJOINT function
        module procedure MatTrace
        end interface
```

This is ending of the code dedicated to the module *matrices*. It si worth noticing that it is not possible to use the command $stop$, differently from $program$.

```
        !no stop for modules
        end module matrices
```

After a MODULE, it is necessary to have a PROGRAM, which is here called *derived_type*. In that module, there are defined two type doublecomplex_matrix: $matrix$ and $matrix1$. These matrixes are initialized using the subroutine initializ_matrices, which initializes the type doublecomplex_matrix, with imput the matrix, its number of columns and its number of rows. The second matrix has many rows as the columns of the first one, and many columns as the rows of the first one.
The second matrix becomes the adjoint of the first one via the function MatAdjoint, interfaced with .Adj. . If squared, the entries of the type doublecomplex_matrix of both matrices dedicated to the trace are filled using the function MatTrace, interfaced with .Tr. . If rectangular, a warning message is printed on the terminal, thanks to the MatTrace function. In order to be clear, another printed message recalls that the warning messages refers to the imput matrix and its adjoint. Lastly it is called print_matrices for both matrices. The first is printed on a file called "DCMatrix.txt", the second on a file called "AdjMatrix.txt".

```
        program derived_type
        use matrices

        integer*2 :: numrow, numcol
        type(doublecomplex_matrix) :: matrix
        type(doublecomplex_matrix) :: matrix1
        !This program defines a type doublecomplex_matrix.
        !It is defined another matrix, which will become the
        !adjoint of the first one.
```

```
numrow=2
numcol=2

call initializ_matrices (matrix, numrow, numcol)
call initializ_matrices (matrix1, numcol, numrow)
!Here it is called calls a subroutine which initializes
!the two routines, with imput the matrix, its number of
!columns and its number of rows.


matrix1=.Adj.(matrix)
!The second matrix becomes the adjoint of the first one
!via the function MatAdjoint, interfaced with .Adj. .

if (abs(DD%nr-DD%nc) < 0.5) then
        print*, "Imput␣matrix␣and␣its␣adjoint:"
end if
matrix%m_trace= .Tr.(matrix)
matrix1%m_trace= .Tr.(matrix1)
!The entries of the type doublecomplex_matrix of both matrices
!dedicated to the trace are filled using the function MatTrace,
!interfaced with .Tr. (if the matrix is squared, otherwise a warn

call print_matrices(matrix, "DCMatrix.txt")
!Here it is called a function which prints the first
!type doublecomplex_matrix on a file, called "DCMatrix.txt".
call call print_matrices_adj(matrix1, "AdjMatrix.txt")
!Here it is called a function which prints the second
!type doublecomplex_matrix on a file, called "AdjMatrix.txt".


stop
end program derived_type
```

Determinant

We implement external functions in order to compute the determinant. The function *determinant*, calling the external function $zgetrf$, is below. Note that it is necessary to convert the imput integer*2 NN in one integer*4 NN_fun because the externl function needed that. $zgetrf$ computes the eigenvaluesof the matrix, then the determinant is obtained by the product of all those.

```
function determinant (NN, mmatrix) result (dett)

integer*2, intent(in) :: NN
type(doublecomplex_matrix), intent(in) :: mmatrix
```

```
integer*4 :: NN_fun
integer :: ii, iinfo

integer, allocatable :: ipiv(:)

real :: sgn

double complex :: dett

allocate(ipiv(NN))

NN_fun = NN

ipiv = 0


call zgetrf(NN_fun, NN_fun, mmatrix%elements,
$                        NN_fun , ipiv, iinfo)


dett = complex(1d0,1d0)

do ii = 1, NN

dett = dett*mmatrix%elements(ii, ii)

end do

sgn = 1

do ii = 1, NN

if(ipiv(ii) /= ii) then

sgn = -sgn

end if

end do

dett = sgn*dett

end function determinant
```

That function is called within the program with:

```
matrix%m_det  = determinant(matrix%nr, matrix)
```

Due to the fact that $zgetrf$ needed several other external functions, it is necessary to link all the needed external function (obtained in LAPACK) in the compilation command:
gfortran ex2_umt.f xerbla.f ilaenv.f zgetf2.f zgemm.f zlaswp.f ztrsm.f ieeeck.f iparmq.f dlamch.f izamax.f zswap.f zscal.f zgeru.f lsame.f dcabs1.f zgetrf.f disnan.f dlaisnan.f

Results, without determinant

Square Matrix, without determinant

The following results are given for a 2×2 matrix, before it was added the code dedicated to the comptation of determinant. The choice of the 2×2 was due to the purpose to show the results on Latex, preserving the proper order of rows and columns. Everything works: the initialization, the computation of the trace and of the adjoint and the printing. A comment must be done on the determinant value of the second matrix. It should be zero, as inizialized, but it is different from a zero for a very small quantity, of the order of 10^{-310}. This is due to the fact that the computer is not infinitely precise. This was announced by the following note after the execution of the program: **Note: The following floating-point exceptions are signalling: IEEE_DENORMAL**.

IMPUT MATRIX:
NUMBER OF ROWS: 2
NUMBER OF COLUMNS: 2
ELEMENTS:
(1.0000000000000000,1.0000000000000000) (1.0000000000000000,2.0000000000000000)
(2.0000000000000000,1.0000000000000000) (2.0000000000000000,2.0000000000000000)
TRACE: (3.0000000000000000,3.0000000000000000)
DET: (0.0000000000000000,0.0000000000000000)

ADJOINT MATRIX:
NUMBER OF ROWS: 2
NUMBER OF COLUMNS: 2
ELEMENTS:
(1.0000000000000000,-1.0000000000000000) (2.0000000000000000,-1.0000000000000000)
(1.0000000000000000,-2.0000000000000000) (2.0000000000000000,-2.0000000000000000)
TRACE: (3.0000000000000000,-3.0000000000000000)
DET: (6.95257086758782842E-310,4.66339582492695348E-310)

Rectangular Matrix

The following results are given for a 2×1 matrix, before it was added the code dedicated to the comptation of determinant. The choice of the 2×1 was due to the purpose to show the results on Latex, preserving the proper order of rows and columns. The trace is a complex

zero as it should be. The determinant differs from zero for the adjoint matrix, due to computers' innate imprecision (like before).
IMPUT MATRIX:
NUMBER OF ROWS: 2
NUMBER OF COLUMNS: 2
ELEMENTS:
(1.0000000000000000,1.0000000000000000)
(2.0000000000000000,1.0000000000000000)
TRACE: (0.0000000000000000,0.0000000000000000)
DET: (0.0000000000000000,0.0000000000000000)

ADJOINT MATRIX:
NUMBER OF ROWS: 2
NUMBER OF COLUMNS: 2
ELEMENTS:
(1.0000000000000000,-1.0000000000000000) (2.0000000000000000,-1.0000000000000000)
TRACE: (0.0000000000000000,-0.0000000000000000)
DET: (6.95308823697660201E-310,4.64074502933092240E-310)

Square Matrix, with determinant

The following results are given for a 2×1 matrix, before it was added the code dedicated to the comptation of determinant. The choice of the 2×1 was due to the purpose to show the results on Latex, preserving the proper order of rows and columns. It is noticeable that the initialization of the matrix do not go well. Also making the adjoint is unsuccessful. The result does not change adding optimization flags.

IMPUT MATRIX:
NUMBER OF ROWS: 2
NUMBER OF COLUMNS: 2
ELEMENTS:
(2.0000000000000000,1.0000000000000000) (2.0000000000000000,2.0000000000000000)
(0.60000000000000009,0.20000000000000001) (0.19999999999999984,0.39999999999999991)
TRACE: (2.1999999999999997,1.3999999999999999)
DET: (0.99999999999999989,-0.99999999999999944)

ADJOINT MATRIX:
NUMBER OF ROWS: 2
NUMBER OF COLUMNS: 2
ELEMENTS:
(1.0000000000000000,-2.0000000000000000) (2.0000000000000000,-2.0000000000000000)
(0.60000000000000009,0.20000000000000001) (0.39999999999999991,-0.19999999999999984)
TRACE: (1.3999999999999999,-2.1999999999999997)
DET: (-0.99999999999999989,0.99999999999999944)

Self-Evaluation

The main goals achieved:

- Defining a new type, with also some elements of that type which will need to be allocated
- Initializating that new type, via some parameters given from the outside (such as the number of rows and columns).
- Printing the elements of the defined type on a file, including the proper printing of a matrix.
- Dealing with functions: defining the variables at the beginning, stating which are imput, allocating the output.
- Computing the trace and the adjoint of a given matrix.
- Writing a check which avoids possible imprecisions of the computer.
- Writing an interface for a function.
- Dealing with modules and programs: the *stop* command is needed only for programs. Moreover, both pieces of codes are needed.
- Layering the structure of the code.
- Using external function: searching them on LAPACK, calling them, linking them.

What was not achieved:

- Combining computation of the determinant with the accuracy of the code. What we can infer is that the computational cost of all those external function is so high that affects all the execution of the code. Even using optimization flags the result does not change.

Complete Code

```
module matrices
implicit none
type doublecomplex_matrix
!defining the type: doublecomplex matrix
integer*2 :: nr
integer*2 :: nc
double complex, dimension (: , :), allocatable :: elements
double complex :: m_trace
double complex :: m_det
end type doublecomplex_matrix
```

```
interface operator (.Adj.)
!.Adj. stands for ADJOINT function
module procedure MatAdjoint
end interface

interface operator (.Tr.)
!.Tr. stands for ADJOINT function
module procedure MatTrace
end interface


contains

subroutine initializ_matrices (AA, numrow, numcol)
!this subroutine initializes a type doublecomplex_matrix,
!given as imput. Also the numbers of rows and columns
!are given as imput. The subroutine creates a matrix
!such that at the entry (i,j) there is the d_complex
!number (i,j). It gives a default value 0 to the
!determinant and to the trace
integer yy, xx
type(doublecomplex_matrix) :: AA
integer*2 numrow, numcol
AA%nr= numrow
AA%nc= numcol
allocate (AA%elements(AA%nr, AA%nc))
do yy= 1, AA%nr
do xx= 1, AA%nc
AA%elements(yy,xx)=complex(1d0*yy,1d0*xx)
end do
end do
AA%m_trace=(0d0,0d0)
AA%m_det=(0d0,0d0)
end subroutine initializ_matrices

subroutine print_matrices (AA, namefile)
!This subroutine prints a type doublecomplex_matrix:
!the numbers of row and columns, the elements
!(in the proper order), the trace and the determinant.
!This is all printed on a file,
!which name is given as imput.
type(doublecomplex_matrix) :: AA
character(:), allocatable :: namefile
integer :: ii,jj
```

```
open(unit = 40, file = namefile, status = "unknown")
write (40,*) "NUMBER␣OF␣ROWS:", AA%nr
write (40,*) "NUMBER␣OF␣COLUMNS:", AA%nc
write (40,*) "ELEMENTS:"
do ii=1, AA%nr !do cycle in order to print in the proper order
write (40,*) (AA%elements(ii, jj), jj = 1, AA%nc)
end do
write (40,*) "TRACE:", AA%m_trace
write (40,*) "DET:", AA%m_det
close (40)

end subroutine print_matrices



function MatAdjoint (CC) result (BB)
!This function, given a type doublecomplex_matrix in imput,
!gives as result a type doublecomplex_matrix which is
!the adjoint of the imput. (number of rows and columns
!are exchanged and the output matrix is the adjoint
!of the imput matrix)
integer :: kk, qq
type(doublecomplex_matrix), intent(in) :: CC
type(doublecomplex_matrix)  :: BB
BB%nr=CC%nc
BB%nc=CC%nr

allocate (BB%elements(BB%nr, BB%nc))

do kk= 1, CC%nr
do qq= 1, CC%nc
BB%elements(qq,kk)=conjg(CC%elements(kk,qq))
end do
end do

end function MatAdjoint

function MatTrace (DD) result (trace)
!This function, given a type doublecomplex_matrix as a imput,
!gives as result the trace of that matrix.
!If it is a square matrix, the trace is computed.
!If it is rectangular, a warning is printed
!and a zero value is returned.
integer :: tt
```

```
double complex trace

type(doublecomplex_matrix), intent(in) :: DD
trace = (0d0, 0d0)
if (abs(DD%nr-DD%nc) < 0.5 ) then !The matrix is squared?
do tt= 1, DD%nr
trace= trace + DD%elements(tt,tt)
end do
else
print*, "WARNING: The matrix is not squared: no trace computed"
end if
end function MatTrace


function determinant (NN, mmatrix) result (dett)

integer*2, intent(in) :: NN
type(doublecomplex_matrix), intent(in) :: mmatrix
integer*4 :: NN_fun
integer :: ii, iinfo

integer, allocatable :: ipiv(:)

real :: sgn

double complex :: dett

allocate(ipiv(NN))

NN_fun = NN

ipiv = 0



call zgetrf(NN_fun, NN_fun, mmatrix%elements,
$                        NN_fun , ipiv, iinfo)



dett = complex(1d0,1d0)

do ii = 1, NN

dett = dett*mmatrix%elements(ii, ii)
```

```
end do

sgn = 1

do ii = 1, NN

if(ipiv(ii) /= ii) then

sgn = -sgn

end if

end do

dett = sgn*dett

end function determinant




c       stop !no stop for modules
end module matrices




program derived_type

use matrices
integer*2 :: numrow, numcol
type(doublecomplex_matrix) :: matrix
type(doublecomplex_matrix) :: matrix1
!This program defines a type doublecomplex_matrix.
!It is defined another matrix, which will become
!the adjoint of the first one.
character(:), allocatable :: firstfile, adjfile

numrow=2
numcol=2
firstfile= "DCMatrix.txt"
```

```
adjfile= "AdjMatrix.txt".

call initializ_matrices (matrix, numrow, numcol)
call initializ_matrices (matrix1, numcol, numrow)
!Here calls a subroutine which initializes the two type,
!with imput the matrix, its number of columns and its number of r

matrix1=.Adj.(matrix)
!Then, the second matrix becomes the adjoint of the first
!one via the function MatAdjoint, interfaced with .Adj. .


if (abs(numrow-numcol) > 0.5 ) then
print*, "Imput␣matrix␣and␣its␣adjoint:"
end if
matrix%m_trace= .Tr.(matrix)
matrix1%m_trace= .Tr.(matrix1)
!The entries of the type doublecomplex_matrix of both matrices
!dedicated to the trace are filled using the fucntion MatTrace,
!interfaced with .Tr. (if the matrix is squared, otherwise
!a warning is printed and the trace is zero).

matrix%m_det  = determinant(matrix%nr, matrix)
matrix1%m_det  = determinant(matrix1%nr, matrix1)


call print_matrices(matrix, firstfile)
!Here it is called a function which prints the first type
!doublecomplex_matrix on a file, called "DCMatrix.txt".
call print_matrices(matrix1, adjfile)
!Here it is called a function which prints the first type
!doublecomplex_matrix on a file, called "AdjMatrix.txt".

stop
end program derived_type
```