Excercise 2

Abstract

This FORTRAN-90 project consists in the definition of a custom type storing square a matrix in the form of a 2d array and some related quantities: trace, determinant and dimension. Then was requested to code functions to initialize the type, to compute trace, adjoint matrix, and to output in a file the instances of the type. Moreover it was necessary to allow the invocation of the functions to compute trace and adjoint using user defined operators.


```
subroutine Init(N, result)
    implicit none

    integer, intent(IN) :: N
    type (matrixqic), intent(OUT) :: result
    !Print*, "Constructor of empty matrixqic.."

    allocate(result%el(N,N))
    result%dim = N
    result%el =0
    result%det =0
    result%trace =0
end subroutine Init
```

Figure 1: Subroutine used to initialize the MATRIXQIC type.

1 Theory

The preliminary notions necessary for the explanation of the content of this exercise are the one of trace and adjoint of a square matrix.

In fact given a square complex matrix $M \in \mathbb{C}^{n,n}$, we define the trace as:

$$tr(M) = \sum_{1}^{N} M_{ii} \tag{1}$$

i.e. the sum of all the elements of the principal diagonal.

While as the adjoint matrix of M:

$$Adj(M) = (M^*)^T \tag{2}$$

i.e. the conjugate transposed matrix of M

2 Code Development

All the code developed has been wrote inside a fortran module MMATRIXQIC. The type has been defined with the name MATRIXQIC, and contains four instances DET (determinant), TRACE (trace) , EL (the whole matrix in form of a 2d array), and DIM (dimension of the matrix).

```
subroutine cgadj(mat, adjmat)
    implicit none
    type(matrixqic), intent(IN) :: mat
    type(matrixqic), intent(OUT) :: adjmat

    integer ii , jj
    double complex, dimension(mat%dim, mat%dim):: tmpmat
    double precision:: im, re

    !Print*, "Computing Adjoint..."

    tmpmat = transpose(mat%el)

    tmpmat = conjg(tmpmat)
    !do ii=1,mat%dim
    !    do jj=1,mat%dim
    !        im =  -aimag(tmpmat(ii,jj))
    !        re =  real(tmpmat(ii,jj))
    !        tmpmat(ii,jj) = cmplx(re, im, kind (0D0))
    !    end do
    !end do
    !call build(mat%dim, tmpmat,adjmat)
    call build_empty(tmpmat,adjmat)
    !Print*,""
    return
end subroutine cgadj

subroutine ctrace(mat)
    implicit none
    type(matrixqic) , intent(INOUT) :: mat
    integer:: ii
    mat%trace=0
    do ii =1, mat%dim
        mat%trace= mat%trace + mat%el(ii,ii)
    end do
end subroutine ctrace
```

Figure 2: Subroutines that compute the adjoint MATRIXQIC and the trace.

2.1 Initialization

To initialize the type a subroutine has been coded, INIT [Fig.1]. This one just allocates memory to store the matrix and fixes the dimension.

2.2 Trace

To compute the trace I coded a subroutine CTRACE[Fig.2] that through a for loop on the diagonal sums all the terms and stores the value in the respective instance of the type.

```
type matrixqic
    integer :: dim
    double complex, dimension(:,:), allocatable :: el
    double complex :: trace
    double complex :: det
end type matrixqic


interface operator(.adj.)
    module procedure adjoint
end interface

interface operator(.tr.)
    module procedure trace
end interface
```

Figure 3: Definition of the type MATRIXQIC declaration of the interfaces.

```
function Adjoint(mat)
    implicit none
    type(matrixqic), intent(IN)::mat
    type(matrixqic) :: adjoint

    call cgadj(mat,adjoint)

end function Adjoint

function Trace(mat)
    implicit none
    type(matrixqic), intent(IN)::mat
    type(matrixqic) :: tmp
    double complex :: trace
    tmp = mat
    call ctrace(tmp)
    trace = tmp%trace

end function Trace
```

Figure 4: Functions called by the interfaces.

```
!output matrixqic in a file
subroutine Ofile(mat,outfile)
    implicit none
    type(matrixqic), intent(IN) :: mat
    character*40, intent(INOUT):: outfile

    character*3 :: stat = "new"
    logical :: file_exist = .FALSE.
    integer :: ii

    outfile = trim(outfile)
    inquire(FILE=outfile, EXIST=file_exist)

    if (file_exist) then
        stat = "old"
    end if

    open(unit = 2, file=outfile, status = stat)

    write(2,*) "dimension = "
    write(2,*) mat%dim
    write(2,*) "trace = "
    write(2, "(*(''sf6.2xspf6.2x'i ':x))") mat%trace
    write(2,*) "determinant = "
    write(2, "(*(''sf6.2xspf6.2x'i ':x))") mat%det
    write(2,*) "full matrix = "


    do ii=1,mat%dim
         write(2, "(*(''sf6.2xspf6.2x'i ':x))")  mat%el(ii,:)
    end do
    close(2)
    return
end subroutine Ofile
```

Figure 5: Subroutine used to output on file.

2.3 Adjoint Matrix

To compute the adjoint I again used a subroutine CGADJ[Fig.2] which returns another MATRIXQIC object, containing in the EL instance the adjoint matrix of the input, TRACE and DET are not computed in the procedure and set to zero.

2.4 Interface Definition

Using the last two subroutines I designed two interfaces, which use the operators .ADJ. and .TRACE.[Fig.1]. The operator .ADJ. is linked then to a function ADJOINT[Fig.4] that calls the CGADJ subroutine, and returns its result. The function linked to the .TRACE. operator

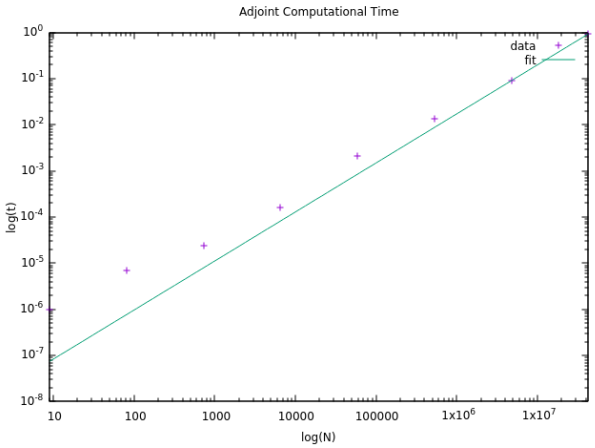

Figure 6: log-log graph representing the computational time took to compute the adjoint of a matrix with N elements. The algorithm order is about $O(n^k)$ with $k \approx 1.06$.

is TRACE which calls the CTRACE subroutine, which sets the TRACE instance of the MATRIXQIC input object, and then returns exactly that value.

2.5 Output on file

To print the data to a file I created a subroutine OFILE[Fig.5] which simply creates a buffer to a file (existing or not), and writes each instance of MATRIXQIC. In particular the EL values are stored in the form $a + bi$, separated by spaces and each row is terminated with a newline.

3 Results

The code showed stability performing tests on matrices filled with random numbers in the range $[0, 1]$. I also studied the computational cost of performing the computation of the adjoint ma-

trix since this task is the most computationally expensive among the requests [Fig.6].

4 Self Development

This code is a simple exercise that exploits the power of subroutines, functions and user defined types. The project can be seen as a good starting point for a simple library containing tools to operate on matrices. Clearly to reach such objective it would be necessary to increase the number of matrix related operations available. Taking in account reliability, low computational cost, and higher abstraction with respect to other existing libraries.