MASTER'S DEGREE IN PHYSICS
Academic Year 2020-2021

QUANTUM INFORMATION

Student: Giorgio Palermo
Student ID: 1238258
Date: November 9, 2020

EXERCISE 5

In this report I am going to review how I solved problems 1 and 2 of Ex. 5, using Fortran functions and subroutines and Gnuplot scripts.

Theory

Mathematically, a Hermitian matrix is a matrix which is equal to its transpose conjugate. The Spectral theorem for finite dimension says that any hermitian matrix can be diagonalized by a unitary matrix; this implie that all eigenvalues of a Hermitian matrix A with dimension n are real, and that A has n independent eigenvectors.

$$H = \begin{bmatrix} 2 & 2+i & 4 \\ 2-i & 3 & i \\ 4 & -i & 1 \end{bmatrix}$$

This, in particular, shows us the equivalence of Hermitian matrices to real diagonal matrices. We are going to use this fact to compare the distribution of distances of contiguous eigenvalues in Hermitian and real diagonal matrices in order to see if they are the same.

Code Development

For this exercise I divided my code in two blocks, one for each problem: `Eigenproblem.f90` and `Statistics.f90`. The purpose of Eigenproblem.f90 is to solve problem 1 generate a dataset for problem 2.

Problem 1 asked to consider an Hermitian matrix, so I chose as fundamental type of my problem the structure `type(dmatrix)` used in previous assignments:

```fortran
type dmatrix
    integer, dimension(2) :: N = (/ 0,0 /)
    double complex, dimension(:,:), allocatable :: elem
    double complex, dimension(:,:), allocatable :: evec
    double precision, dimension(:), allocatable :: eval
    double complex :: Trace
    double complex :: Det
end type dmatrix
```

The procedure to follow in order to create a meaningful dataset for problem two is the following:

1. Matrix initialization
2. matrix diagonalization
3. normalized spacings calculation
4. output to file
5. repeat

Writing this code I wanted to assign as many operation as possible to functions, in order to be able to automatize them efficiently; I also used as many LAPACK functions as I could in order to call the best implementation possible for the various algorithms.

For the first step of the algorithm I implemented `function InitHerm(herm_1,Nmat)`: this function takes as input a `type(dmatrix)` object and initializes it to a Hermitian matrix of size `Nmat`. In particular, all array fields of the structure are allocated, all scalars are initialized to 0 and the `double complex` array containing the matrix is initialized to a random Hermitian matrix using the LAPACK subroutine `zlaghe`.

The second step af the algorithm is performed in a similar way, using

```
function DmatHermEv(herm_1)
```

wich is nothing else than a simple interface between the LAPACK solver `zheev` and the custom `type(dmatrix)`. I used in this program to manage complex matrices; as a matter of fact, `herm_1` is a `type(dmatrix)` data. The eigenvalues of the input matrix are stored in a crescent order in the `herm_1%eval` field of the output.

The last strictly computational step regards computation of normalized spacings between eigenvalues. I included this operation into

```
function SpAvg(herm_1).
```

I computed normalized spacings using the `eoshift` intrinsic function, which shifts all the elements of an array of an arbitrary amount along a specified direction. Subtracting a properly shifted array to the one containing the original eigenvalues gives the spacings S_i, as shown in the code below:

```
function SpAvg(herm_1)
    implicit none
    ! Local custom types
    type(dmatrix) :: herm_1
    ! Local scalars
    integer :: Nmat,sp_size
    double precision :: s_avg, sp_sum
    ! Local arrays
    double precision, dimension(:), allocatable :: values, sp_0, sp
    double precision, dimension(herm_1%N(1)-1) :: SpAvg

    allocate(values(herm_1%N(1)))
    allocate(sp_0(herm_1%N(1)))
    allocate(sp(size(sp_0-1)))
    values = herm_1%eval
    sp_0=values
    values=eoshift(values, shift=-1)
    sp_0 = sp_0-values
    sp=sp_0(2:)
    sp_size = size(sp)
    sp_sum = sum(sp)
    s_avg = sp_sum/sp_size
    sp=sp(:)/s_avg

    SpAvg = sp
end function SpAvg
```

I chose to use this method because it allows not read explicitly all the eigenvalues with a loop in order to compute S_i.

Output and repetition are implemented in the main file `Eigenproblem.f90` itself. This program is essentialy a loop that repeats initialization, diagonalization and output for a fixed matrix size (`Nmat`) `Ncycles` times. This program produces a dataset automatically named after the matrix size and the number of iterations (e.g.`Sp_1000_0100.dat`) which contains in each row all the normalized spacings between the eigenvalues of an hermitian matrix for a `Ncycles` number of rows. A live output of the percentage of completion of the loop is displayed on screen.

The code is the following:

```
! External modules:
include "ModDmat.f90"
include "ModDebug.f90"

program Eigenproblem
    use ModDmat
    use ModDebug
    implicit none
    ! Local scalars
    integer :: Nmat, Nsp, ios, ii,jj, Ncycles
    character(len=100) :: filename, msg, x1,x2, format
    ! Local arrays
    double precision, dimension(:), allocatable :: spacings
```

```fortran
    double precision :: perc, start, end, time
    ! Local custom types
    type(dmatrix) :: herm_1

    Nmat = 1000
    Ncycles=100
    ... Initialization of filename and some parameters ...
    do jj=1,Ncycles
        call cpu_time(start)
        allocate(herm_1%elem(Nmat,Nmat))
        herm_1=herm_1 .Init. Nmat
        herm_1 =.evalh.herm_1
        if (isnan(sum(herm_1%eval))) go to 129
        spacings=SpAvg(herm_1)

        write(x1,'(i4.4)') Nmat
        do ii=1,Nsp
        write(55,'(g13.6)', advance='no') spacings(ii)
        end do
        129 deallocate(herm_1%elem)
        deallocate(herm_1%eval)
        write(55,*)
        call cpu_time(end)
        time=end-start
        perc=100*jj/Ncycles
        write(*,'("Elapsed time [s]: ",(G9.2),(G9.2)," % done...")') time, floor(perc)
    end do
    close(55)
end program Eigenproblem
```

The solution of problem 2 is computed by the program `Statistics.f90` by using the output files from `Eigenproblem.f90`. To make some statistics on the data I chose to define a `type(histogram)` structure. I created this place to store in a compact and ordered way the histograms and all the related informations that I needed for the analysis, but I couldn't fit into a single array:

```fortran
type histogram
    integer :: Nbins, entries
    double precision ::lower, upper
    integer, dimension(:), allocatable :: h
    double precision, dimension(:), allocatable :: hnorm
    double precision, dimension(:), allocatable :: bounds, bincenters
end type histogram
```

`Statistics.f90` basically works on two of these histograms: it loads the data, it fills them and it saves the relevant information onto an automatically named file like `Eigenproblem.f90` did. The first one, `h1`, contains the data retrieved from diagonalization of random Hermitian matrices, while the second one, `h2`, contains data from randomly generated double precision diagonal matrices. The generation of random double precision data is made with the intrinsic subroutine `random_number`; this data is then sorted in a crescent order using LAPACK subroutine `dlasrt` and passed to a appropriately modified `SpAvg` function (`SpAvgDble`) that returns the normalized spacings.

```fortran
! External modules:
include "ModDmat.f90"
include "ModDebug.f90"

program Statistics
    use ModDmat
    use ModDebug
    implicit none
    ! Local scalars
    character(len=100) :: msg, x1, x2, filename, filename_1
    integer :: Nsp, Ncy, ios, Nbins, ii, seme_dim=4
    double precision :: h_lenght, h_step, h_lower, h_upper
    ! Local arrays
    double precision, dimension(:,:), allocatable :: indata
    double precision, dimension(:), allocatable :: diag_rnd, sp_diag
    double precision, dimension(:), allocatable :: h_bounds, h_input_aux
    integer, dimension(:), allocatable :: h_input, h, seme

    ! Local custom types
    type(histogram) :: h1, h2
    ... filename definition ...
    Nsp=1999
    Ncy=450
    h_lower = 0
    h_upper = 8
    Nbins=50

    allocate(indata(Ncy,Nsp),diag_rnd(Nsp+1),sp_diag(Nsp))
    indata = LoadArray("Sp_2000_0450.dat",Ncy,Nsp)

    h1=InitHisto(h_lower,h_upper,Nbins)
    h2=InitHisto(h_lower,h_upper,Nbins)
    do ii=1,Ncy
        call FillH(h1,indata(ii,:),"y")
    end do

    call random_seed()
    call random_seed(size=seme_dim)
    allocate(seme(seme_dim))
    call random_seed(get=seme)
    call random_number(diag_rnd)

    call dlasrt('I',Nsp+1,diag_rnd,ios)
    sp_diag=SpAvgDble(diag_rnd)

    call FillH(h2,sp_diag,'y')

    call HistoToFile(h1,filename)
    call HistoToFile(h2,filename_1)

end program Statistics
```

Histograms are filled by `subroutine FillH(h1,indata,norm)`. This function uses an `intent(inout)` argument `type(histogram) :: h` (previously initialized by a specific function), an argument for input data `indata` and a flag that specifies if a normalized histogram has to be saved or not `norm`. Using `FillH`, `indata` is sort into `h%Nbins` categories comprised into bounds defined in `h%bounds` vector; a vector with the center of the bins is also saved for plotting purposes.

```fortran
subroutine FillH(h1,indata,norm)
    implicit none
    ! Local scalars
    integer ::  Nentries
    double precision :: step
    character(*),intent(in) :: norm
    ! Local arrays
    double precision, dimension(:), intent(in) :: indata
    double precision, dimension(:), allocatable :: h_input_aux
    integer, dimension(:), allocatable :: h_input

    ! Local custom types
    type(histogram),intent(inout) :: h1

    step = (h1%upper - h1%lower)/h1%Nbins
    Nentries=size(indata)
    allocate(h_input_aux(Nentries),h_input(Nentries))
    h_input_aux= indata
    h_input_aux = ceiling(h_input_aux/step)
    h_input=int(h_input_aux)
    h1%h(h_input) = h1%h(h_input)+1
    h1%entries = sum(h1%h)
    if(norm=="y".and. h1%entries/=0) then
          h1%hnorm=dble(h1%h)/h1%entries*step
    end if
end subroutine FillH
```

In the end, `Statistics.f90` gives as output a text file for each histogram, containing center coordinates and normalized values for each bin.

Results and self evaluation

To retrieve the fit parameters requested by problem 2, I chose to generate a dataset of normalized eigenvalue spacings retrieved from 2000×2000 matrices; I set the `Eigenproblem.f90` cycles counter to

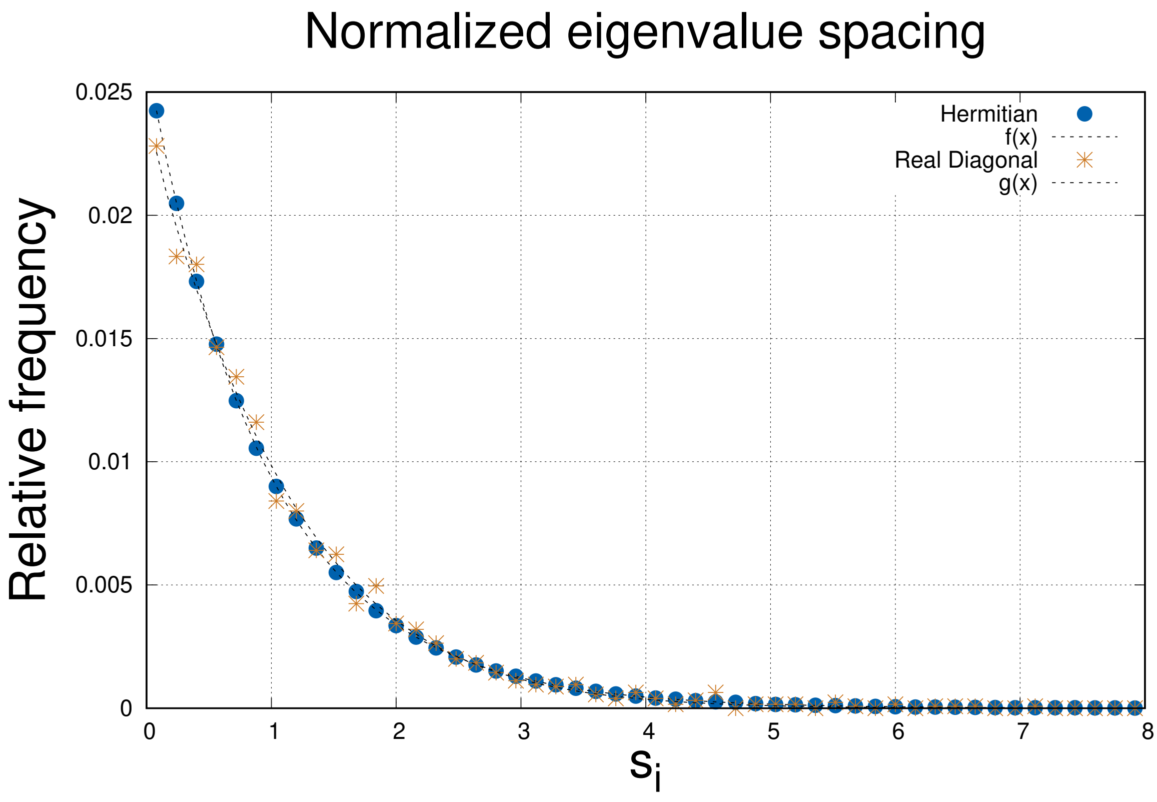

Figure 1: Normalized eigenvalue spacings from random Hermitian matrices (blue) and from random real diagonal matrices (orange).

450 to generate approximately 9×10^5 data points (normalized spacings). I filled an histogram with these data, which can be seen in figure 1. For both histograms I set $Nbins = 50$ and a binning interval spanning from 0 to 8 (dimensionless); this produces little dispersion of data, of the order of 10^2 points, which is negligible.

The data retrieved from Fortran programs as described above has been fitted to the law

$$f(x) = ax^\alpha \exp\left(bx^\beta\right)$$

using gnuplot. Results for a, α, b, β are the following for the two datasets:

Table 1: Fit results for Hermitian (first row) and real diagonal (second row) matrices.

	$\boldsymbol{a}$	$\boldsymbol{\alpha}$	$\boldsymbol{b}$	$\boldsymbol{\beta}$	$\boldsymbol{\chi^2}$**/d.o.f.**
$f(x)$	0.0263 ± 0.0002	-0.001 ± 0.003	1.033 ± 0.008	0.987 ± 0.002	1.6×10^{-8}
$g(x)$	0.021 ± 0.002	-0.04 ± 0.04	0.8 ± 0.1	$1.2 \pm 0.1 \pm 0.005$	3×10^{-4}

As a conclusion we can observe that some parameters of the two distributions are not compatible and therefore we cannot state that they are the same. Nonetheless the similarity between the two is remarkable and it is a clear sign of correlation, which might be confirmed by more specific analysis.