# Quantum information and Computing
# Ex. 05

Andrea Nicolai - 1233407

November 9, 2020

**Abstract**

In this work we will compute the spacings between eigenvalues of both Hermitian and Diagonal matrices. These spacings are then normalized either wrt their global or local average. A fit is done in order to compute the distribution of these spacings for different choices of parameters and finally is compared to the theoretical curves.

## Theory

Hermitian matrices are also known as self-adjoint matrices: they are complex matrices equal to their transposed and conjugated. In this work we are going to study the behavior of their set of ordered eigenvalues $\{\lambda_i\}_{i=1,N}$, where $\lambda_1 < \lambda_2 < ... < \lambda_N$, keeping distinct whether we are either considering general Hermitian matrices "$H$", or Diagonal matrices "$D$" whose entries are real. Moreover we know from spectral theorem that for a matrix of these kinds all its eigenvalues are real. Specially we want to study the normalized spacings between each element in the set of ordered eigenvalues:

$$s_i = \frac{\lambda_{i+1} - \lambda_i}{\bar{\Delta\lambda}} \tag{1}$$

Where the "normalization" term $\bar{\Delta\lambda}$ might be either *general*, namely the average value over of all $\lambda$'s, or *local* if we consider the average over only a window whose size is to be defined.
From theory we know they distribute according to the Wigner quasi-probability distribution:

$$P(s) = 2\left(\frac{4s}{\pi}\right)^2 exp\left(\frac{4s^2}{\pi}\right) \tag{2}$$

We should moreover keep into account the specific case regarding Diagonal matrices, that makes the distribution collapse to an exponential one: $P(s) = e^{-s}$. Lastly, we are asked to compute also the average $< r >$ for each choice of parameter we may set, where $r_i$ is:

$$r_i = \frac{MIN(\Delta\lambda_{i+1}, \Delta\lambda_i)}{MAX(\Delta\lambda_{i+1}, \Delta\lambda_i)} \tag{3}$$

## Code Development

In first place we used a code we had already implemented in order to initialize a double complex entries matrix. We had to take into account that it had to be Hermitian: after drawing uniformly both real and imaginary coefficients in [-1,+1], we introduced a check control whether two symmetric entries were one the conjugate of the other. Once it was successful, we needed to get eigenvalues. Therefore, we had to install *llapack* package for Fortran and use "-llapack" flag when compiling, so we would be able to use the following subroutine that accepts, as most important quantities for our case, 2-dim double complex array and a double array which will be filled with the eigenvalues ordered increasingly. Note as, since from theory we know that eigenvalues of a Hermitian matrix are real, the array is of $REAL*8$ type and not $DOUBLE\ COMPLEX$ as it is for the entries of the matrix. The following code exploits $ZHEEV$ llapack subroutine in order to obtain what we are looking for:

```fortran
SUBROUTINE HERMITIAN_DIAGONALIZE(matrix, matrix_size, &
& array_eigenvalues, info)
   (...)
   !Some useful variables
   DOUBLE COMPLEX, DIMENSION(:,:), INTENT(INOUT) :: matrix
   REAL*8, DIMENSION(:), ALLOCATABLE, &
   & INTENT(INOUT) :: array_eigenvalues
   (...)
   !if -1, then we call subroutine in order to obtain the &
   &optimal lwork value
   lwork = -1
   ALLOCATE(work(MAX(1, lwork)))
   ALLOCATE(rwork(MAX(1, 3*matrix_size-2)))
   (...)
   CALL ZHEEV('N', 'U', matrix_size, matrix, matrix_size, &
   array_eigenvalues, work, lwork, rwork, info)
   lwork = int(real(work(1)))
   DEALLOCATE(work)
   !this call actually does the "dirty job"
   ALLOCATE(work(MAX(1,lwork)))
   CALL ZHEEV('N', 'U', matrix_size, matrix, matrix_size, &
   & array_eigenvalues, work, lwork, rwork, info)
   (...)
   CALL SHELL_SORT_ARRAY( array_eigenvalues )
END SUBROUTINE
```

In last place note that we call the *shell sort* algorithm subroutine in order to return the array sorted increasingly, as requested. We choose not to show the code for it, for a matter of space.
Once sorted the array, we call the following subroutine to compute the spacings between every couple of them. Only the declaration here is shown.

```fortran
SUBROUTINE DELTA_ELEMENTS_ARRAY(array, delta_array)
```

Once obtained the different spacings, there could be a call to a subroutine that returns the array whose elements are averaged either globally or locally, depending on the parameter we passed as an argument to the program call. If it were 0, then the subroutine called would be:

```
SUBROUTINE NORMALIZED_SPACINGS_TOTAL(array, norm_array)
```

that would return the average over all eigenvalues. Otherwise, the integer passed as argument would be the ”window” size when computing the local average, i.e. the number of elements before and after the *i*-th spacing taken into account.

```
SUBROUTINE NORMALIZED_SPACINGS_LOCAL(array, &
& norm_array, window_size_side)
```

Last function we implemented when dealing with the $\Delta\lambda$ arrays is the following, that computes the average value of the $r$ term we introduced in the theory paragraph.

```
FUNCTION COMPUTE_R_MEAN(delta_array) RESULT(r_mean)
```

We were then requested to study the distribution of $P(s)$ of the normalized spacings. In order to do so, we first took a look at our output data and, once chosen an interval $[0, 3.5]$ that would contain around 99% of all $s_i$ values, we defined a new derived type named *Histogram* that would help in our purpose. Its main feature was to split the aforementioned interval into a certain number of bins (*n_bins*) which we set to be 350, and compute how many events (*counts*) fall within a certain bin defined by its edges (*edges*).

```
TYPE HISTOGRAM
    REAL*8 :: upper, lower
    INTEGER :: n_bins
    REAL*8,  DIMENSION(:), ALLOCATABLE  :: edges, bins
    !Following values have to be filled
    REAL*8,  DIMENSION(:), ALLOCATABLE  :: densities
    REAL*8                              :: bin_width
    INTEGER, DIMENSION(:), ALLOCATABLE  :: counts
END TYPE HISTOGRAM
```

Later, we would sum up all the elements of *counts* array in order to obtain the *density* column, by normalizing. In this way we obtain an histogram object whose columns all sum up to 1. Actually, despite the name, we will not use them to compute the $P(s)$, which is a probability density function, since we still need to normalize wrt the sum of the *densities* array entries multiplied by the *bin_width*. This is done by calling the following subroutine:

```
SUBROUTINE OBTAIN_PDF(hist)
IMPLICIT NONE
   TYPE(HISTOGRAM), INTENT(INOUT) :: hist
   (...)
   norm_factor = SUM(hist%counts*hist%bin_width)
   DO ii = 1, hist%n_bins
      hist%densities(ii) = hist%counts(ii)/norm_factor
   END DO
   (...)
END SUBROUTINE
```

As one can see, it replaces the *densities* entries with the actual *densities* that defines the pdf. A debug print was written moreover to check whether the area was correctly 1.
These histograms were therefore to be written with the following format (*bin center*; *density*) to an output file that was in turn fed to a gnuplot script that computed the proper fit script. The function whose parameters were to be estimated was the following:

$$P(s) = as^{\alpha}e^{-ns^{\beta}} \tag{4}$$

and the values for $a, b, \alpha, \beta$, estimated by the mean of MSE algorithm, are finally written as output onto a *csv* that keeps track of them.
We moreover wrote a Python script that automatizes both the fits procedure and the kind of matrices to be implemented, being them either Hermitian ($H$) or Diagonal ($D$), their size (*matrix_size*), how many samples we need to generate (*numb_matrices*) and finally the *moving_param*, whose meaning we pointed out before. If one of these 4 arguments were missing, then the program would raise an error in order to obtain all of them together.

## Results

We run the program initializing 200 samples of $1500 \times 1500$, $H, D$, and with different sets of moving parameters: $(0, 1, 15, 30, 150)$ we obtain the following results for the fit for Hermitian matrices:

| **Window size param** | **a** | **b** | $\alpha$ | $\beta$ | $<$ **r** $>$ |
|---|---|---|---|---|---|
| 0 (glob. av.) | 12.68 | 2.73 | 2.54 | 1.35 | 0.60 |
| 1 (loc. av.) | 2.37 | 0.94 | 1.84 | 2.34 | 0.60 |
| 15 (loc. av.) | 3.11 | 1.24 | 1.95 | 2.01 | 0.60 |
| 30 (loc. av.) | 3.15 | 1.25 | 1.95 | 2.00 | 0.60 |
| 150 (loc. av.) | 3.68 | 1.42 | 2.03 | 1.88 | 0.60 |

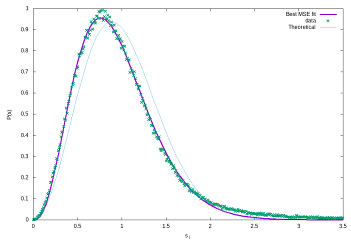
Instead, for the Diagonal matrices we obtain:

| **Window size param** | **a** | **b** | $\alpha$ | $\beta$ | $<$ **r** $>$ |
|---|---|---|---|---|---|
| 0 (glob. av.) | 1.04 | 1.00 | 0.00 | 1.00 | 0.36 |
| 1 (loc. av.) | 0.54 | 0.16 | -0.06 | 2.52 | 0.36 |
| 15 (loc. av.) | 0.97 | 0.93 | -0.00 | 1.04 | 0.36 |
| 30 (loc. av.) | 1.00 | 0.96 | -0.00 | 1.02 | 0.36 |
| 150 (loc. av.) | 1.02 | 0.99 | 0.00 | 1.01 | 0.36 |

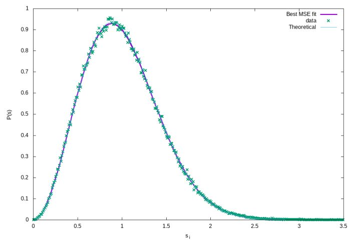
Where the expected values were:

| **Matrix Type** | **a** | **b** | $\alpha$ | $\beta$ |
|---|---|---|---|---|
| H | 3.24 | 1.27 | 2 | 2 |
| D | 1 | 1 | 0 | -1 |

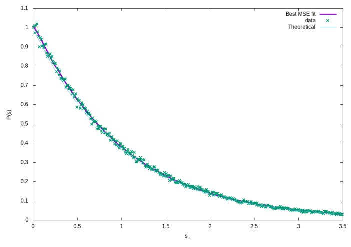
As one can see from the tables, using the local average taking into account the previous and following 30 values we obtain better results, specially if with window size 30. One should note that $< r >$ is constant for $H$ and $D$, despite the choice of parameters for the program.



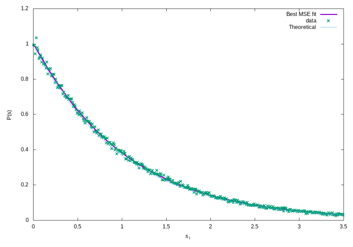
(a) **Hermitian** matrices results, using the **global** average. Fit is not so good.



(b) **Hermitian** matrices results, using the **local** average, i.e. window size is set to 30. This is indeed a very good fitting result.



(c) **Diagonal** matrices result, using the **local** average with window 1. This is the worst fit we obtained for Diagonal matrices!



(d) **Diagonal** matrices result, using the **local** average (window size is set to 30. Fit is really good!

Figure 1: Fitting curves for 2000 samples of $1500 \times 1500$ and different kinds of matrices superimposed to their expected distributions.

## Self Evaluation

We have learnt how to computationally obtain a pdf without using already implemented functions, in order to do that we implemented a new type ($HISTOGRAM$) and defined some operators to deal with it. This was done in order to make the code more elegant and with less redundancy, therefore many modules were implemented. Some more operators, beside the sum one, might be implemented next.