Midterm II

Paul Fischer
Department of Physics
California State University Long Beach

April 20, 2020

Abstract

In Problem 1, we write a `Fortran 90` code that uses the linear algebra package `LAPACK` to explore the physics of the quantum harmonic oscillator. We are given a Hilbert-space basis $\{|n\rangle\}$, $n = 0, 1, 2, ...$, with a lowering operator $\hat{a}$ such that $\langle n'|\hat{a}|n\rangle = \sqrt{n}\delta_{n',n-1}$. We consider $\hat{a}$ in a finite subset of the basis and then determine the corresponding raising operator $\hat{a}^\dagger$. We then determine the position operator $\hat{X} = \sqrt{\frac{\hbar}{2m\omega}}(\hat{a} + \hat{a}^\dagger)$ and momentum operator $\hat{P} = i\sqrt{\frac{m\hbar\omega}{2}}(\hat{a}^\dagger - \hat{a})$ with $\hbar = 1$, $m = 1$, $\omega = 1$. We then determine the Hamiltonian operator $\hat{H} = \frac{\hat{P}^2}{2M} + \frac{1}{2}M\Omega^2\hat{X}^2$ in a 20×20 matrix representation with $M = 2$, $\Omega = \sqrt{2}$. We determine the eigenvalues and eigenvectors of $\hat{H}$ and show that the eigenvectors are orthogonal. Then we use the eigenvectors and eigenvalues to build up the spectral representation of the unit matrix and the $\hat{H}$ matrix. Finally, we construct the function $D(E) = |E - H|$ and show that the eigenvalues coincide with the zeros of the determinant.

In problem 2, we write a `Fortran 90` code utilizing the Newton-Raphson method for locating the zeros and poles of $f(x) = (x\pi)^2 \cot(\pi x)$ for $x \in [0, 5]$ and determine the corresponding residua of the poles.

1 Introduction

At the beginning of the 20th century, theoretical physicists were confronted with having to explain the observation of quantized atomic spectra. Werner

Heisenberg successfully modeled this phenomenon in 1925 with his matrix mechanics. This evolved into the modern formulation for quantum mechanics which fundamentally changed how classical variables were viewed. Position and momentum were then treated as eigenvalues, also called "observables" in this context, of linear operators on a Hilbert space, instead of values of functions on a classical phase space [1]. Determining a solution to a quantum mechanical system can be difficult analytically, but this matrix formalism allows us to use the linear algebra package `LAPACK` to numerically approximate a solution to a quantum mechanical system with a `Fortran 90` code.

Physicists will often want to know the zeros of a function to solve for quantities like, for example, the equilibrium points of a chaotic system. One way to numerically approximate the zeros of a function is by using the multidimensional Newton-Raphson method. If $\mathbf{x}$ is close to the zero we are searching for, we can take the 1$^{\text{st}}$-order term from the Taylor series expanded around that point of $\mathbf{F}(\mathbf{x}+\mathbf{dx})$ and define a Jacobian matrix $\mathbf{J} = \nabla\mathbf{F}$. This leads to the equation $\mathbf{J}(\mathbf{x}_n)\cdot d\mathbf{x} = -\mathbf{F}(\mathbf{x}_n)$, where $\mathbf{x}_{n+1} = \mathbf{x}_n + d\mathbf{x}$ can be used to iteratively determine the multivariable root. The drawback of this method is that it requires a good initial guess [2].

This midterm will be organized as follows. Section *Problem 1* will introduce, show, and provide the output for the code written to solve problem 1. Section *Problem 2* will introduce, show, and provide the output for the code written to solve problem 2. Finally, Section *Summary and conclusions* will provide final thoughts on what we learned in the process of completing this midterm.

2 Problem 1

The `Makefile` used for problem 1 can be found below in Listing 1. It gives instructions for the `gfortran` compiler on how to compile the `Fortran 90` module `numtype.f90` and code `hrmosc.f90` into object files and link them with the library `Accelerate`, which includes the linear algebra package `LAPACK`, in an executable `hrm`. The code can be compiled by typing `make` into the terminal when in the directory `~/src` and executed by typing `hrm`. Flags have been added to the compiling instructions for optimization. By typing `make clean` into the terminal the executable, object, `*.mod`, and `fort.*` files will be removed from the directory leaving only the Makefile and .f90 files required to run the code again from scratch.

Listing 1: `Makefile`

```

objs1 = numtype.o hrmosc.o

prog1 = hrm

f90 = gfortran

f90flags = -O3 -funroll-loops -ftree-vectorize \
    -fexternal-blas

libs = -framework Accelerate

ldflags = $(libs)

all: $(prog1)

$(prog1): $(objs1)
    $(f90) $(ldflags) -o $@ $(objs1)

clean:
    rm -f $(prog1) *.{o,mod} fort.*

.suffixes: $(suffixes) .f90

%.o: %.f90
    $(f90) $(f90flags) -c $<
```

The module `numtype` can be found below in Listing 2. Included in this module are the desired precision of real numbers `dp`, the value of the constant π, numbers that can be multiplied by real data types to turn them into complex data types with precision `dp` as needed, and a real parameter `tiny`.

Listing 2: `Fortran 90` Module `numtype.f90`

```

module numtype

    save
```

```fortran
    integer, parameter :: dp = &
        & selected_real_kind(15,307)
    real(dp), parameter :: pi = 4*atan(1._dp)
    complex(dp), parameter :: z0 = (0._dp, 0._dp), &
        z1 = (1._dp, 0._dp), zi = (0._dp,1._dp)
    real(dp), parameter :: tiny = 1.e-30_dp

end module numtype
```

The code `hrmosc.f90` can be found below in Listing 3. The code begins with the option to enter the number of dimensions desired in the the problem `ndim` for the linear operator matrices. Then we have the option to set the values for the masses, $m :=$ `m` and $M :=$ `Mass`, frequencies, $\omega :=$ `w` and $\Omega :=$ `Omega`, and Planck's constant $\hbar :=$ `hbar`. The precision to which we feel confident equating a numerically approximated quantity to an analytic one `eps` can also be entered.

We use an external function `delta` which represents the Dirac delta, δ, to create and print the lowering operator matrix, $\hat{a} :=$ `a`, in a finite subset of the basis, `ndim`. Then we create the raising operator matrix, $\hat{a}^\dagger :=$ `adag`, using the built-in `Fortran 90` functions `transpose` and `conjg` on the lowering operator, `a`. We create the position and momentum operators, $\hat{X} :=$ `X` and $\hat{P} :=$ `P`, with the information given in the problem. We create the Hamiltonian operator matrix, $\hat{H} :=$ `H`, keeping in mind that $\hat{P}^2 = \hat{P}^\dagger\hat{P}$ (since $\hat{P}$ is always Hermitian, $\hat{P}^\dagger = \hat{P}$, so the built-in matrix multiplication function `matmul` used in the code is sufficient).

The subroutine `zheev` from the package `LAPACK` is used to output the eigenvalues and eigenvectors of the Hamiltonian operator `H` in an eigenvalue array `eval` and matrix `evec` whose n^{th} column is the eigenvector corresponding to the n^{th} eigenvalue of the array `eval`. The eigenvalues are then shown to be orthogonal if `orth`, the multiplication of the eigenvector matrix `evec` with its own conjugate transpose, returns the identity matrix. An `if` block checks if the difference between any element of `orth` and the identity matrix is less than `eps`. The identity matrix used for this comparison is created using the `delta` function.

The spectral theorem is concerned with a class of linear operator matrices $A := UDU^*$, where U is the eigenvector matrix of A and D is the diagonal matrix whose elements are the eigenvalues of A [4]. The spectral representation of the unit matrix in this code is built up by summing the outer products of each of the eigenvectors of `H`. The spectral representation `Hspec` of `H` is the result of a matrix multiplication of `spec` with the conjugate transpose of the eigenvector matrix, where `spec` is the spectral decomposition of `H` obtained by matrix multiplying the transpose of `eval` with the eigenvector matrix.

The last part of the code verifies that the determinant $D(E) = |E - H|$, where E is the product of an eigenvalue and the identity matrix and H is the Hamiltonian operator matrix, should be zero at each eigenvalue E. This was attempted by constructing the function `det`. This function finds `mat` $:= E - H$ for the inputted `E` and `H` values and first determines if the matrix is diagonal. If it is, it skips the the step of checking if there are zeros on the diagonal. Otherwise, it calls the subroutine `zgetrf` to return the LU facortizaion of `mat`. Then it tries to find any zeros on the diagonal which would make the determinant zero. Due to the numerical approximation nature of `Fortran 90`, we discovered that going about solving the determinant for large `ndim` could result in numbers that blow up too large for the code to retain its accuracy. This meant having to cheat in a sense to see if there are any values less than `eps` on the diagonal and say that we're confident they should be zero. That is why the ordering from smallest on the diagonal to largest was done, but it was not enough. The function ends with getting the right sign of the determinant `s`. Since the numbers blew up too large and we had to cheat, we decided to replace this function we constructed in the code with the `LINPACK` subroutine `zgbdi`, which returns the complex determinants for complex matrices and then compares the results with `eps` to show that the eigenvalues coincide with the zeros of $D(E)$. For these linear operators, the eigenvalues and thusly the determinants will be real, so only the real part of them need to be considered; however, the imaginary part is printed as well so that there is no loss of generality.

Listing 3: `Fortran 90` Code `hrmosc.f90`

```

program hrmosc

    use numtype
    implicit none

    ! enter dimension 'ndim' of linear operator matrices
    integer, parameter :: ndim = 20, lwork = 2 * ndim - 1
    integer, external :: delta
    integer :: i, j, k, info, ipiv(ndim)
    real(dp), parameter :: hbar = 1._dp, m = 1._dp, &
        w = 1._dp, Mass = 2._dp, &
        Omega = sqrt(2._dp), eps = 1e-10_dp
    real(dp), external :: det
    real(dp) :: eval(ndim), rwork(3 * ndim - 2), re, &
        im, sum
    complex(dp), dimension(ndim,ndim) :: a, adag, &
        X, P, H, evec, orth, iden, spec, Hspec, &
        idenspec, evecdag, E
    complex(dp) :: work(lwork), D(ndim), determinant(2)

    print *, '-------------------------'

    ! consider lowering operator 'a' in a finite subset
    ! of the basis
    forall(i = 0 : ndim - 1, j = 0 : ndim - 1) &
        & a(i+1,j+1) = sqrt(1._dp * j) * delta(i, j-1)

    print *, 'lowering␣operator␣''a''␣='
    do k = 1, ndim
        ! 'print' format must start with minimum 2*ndim
        print '(40f10.4)', a(k, 1:ndim)
    end do

    print *, '-------------------------'

    ! determine raising operator 'adag'
    adag = transpose(conjg(a))

    print *, 'raising␣operator␣''adag''␣='
    do k = 1, ndim
        print '(40f10.4)', adag(k, 1:ndim)
    end do

    print *, '-------------------------'

    ! determine position operator 'X'
    X = sqrt( hbar / (2 * m * w) ) * (a + adag)

    print *, 'position␣operator␣''X''␣='
    do k = 1, ndim
        print '(40f10.4)', X(k, 1:ndim)
    end do

    print *, '------------------------'

    ! determine momentum operator 'P'
    P = zi * sqrt(m * hbar * w / 2) * (adag - a)

    print *, 'momentum␣operator␣''P''␣='
    do k = 1, ndim
        print '(40f10.4)', P(k, 1:ndim)
    end do

    print *, '------------------------'

    ! determine Hamiltonian operator 'H'
    H = matmul(P, P) / (2 * Mass) &
        & + 1._dp / 2 * Mass * Omega**2 &
        & * matmul(X, X)

    print *, 'Hamiltonian␣operator␣''H''␣='
    do k = 1, ndim
        print '(40f10.4)', H(k, 1:ndim)
    end do

    print *, '------------------------'

    ! determine the eigenvalues and eigenvectors of
    ! Hamiltonian operator 'H'
    info = 0
    evec = H

    call zheev('v', 'u', ndim, evec, ndim, eval, &
        work, lwork, rwork, info)
        if(info /= 0) stop '␣zheev␣info␣/=␣0'

    print *, 'eigenvalue␣array␣''eval''␣='
    do i= 1, ndim
        print '(f10.4)', eval(i)
    end do

    print *, '-------------------------'

    print *, 'the␣eigenvectors␣are␣the␣columns␣of␣''evec''␣='
    do i= 1, ndim
        print '(40f10.4)', evec(1:ndim,i)
    end do

    print *, 'evec(1:ndim,i)␣is␣the␣corresponding', &
        '␣eigenvector␣to␣the␣eigenvalue␣eval(i)'

    print *, '-------------------------'

    ! show that the eigenvectors are orthogonal
    print *, 'the␣eigenvectors␣are␣orthogonal␣if␣the', &
        '␣following␣is␣the␣identity␣matrix:'

    orth = matmul(conjg( transpose(evec) ), evec)

    do i= 1, ndim
        print '(40f10.4)', orth(1:ndim, i)
    end do

        ! print whether or not any element of
        ! 'orth' matrix differs from identity
        ! matrix 'iden' by less than parameter 'eps'
    do i = 1, ndim
        do j = 1, ndim
            iden(i, j) = delta(i, j)
        end do
    end do

    do i = 1, ndim
        do j = 1, ndim
            re = realpart(orth(i,j)) - realpart(iden(i,j))
            im = imagpart(orth(i,j)) - imagpart(iden(i,j))
            sum = abs(re) + abs(im)

            if ( sum > eps) then
```

```
                print *, 'the␣eigenvectors␣are␣not␣orthogonal'
                go to 10
            end if
        end do
    end do
    10 if (i == ndim + 1) then
        print *, 'the␣eigenvectors␣are␣orthogonal'
    end if

    print *, '------------------------'

    ! build up the spectral representation of the unit
    ! matrix 'idenspec' as the sum of the outer products
    ! of the eigenvectors of the Hamiltonian operator 'H'

    forall(i = 1:ndim, j = 1:ndim) &
        & spec(i, j) = evec(i, j) * eval(j)

    evecdag = transpose(conjg(evec))

    do k=1,ndim
        do i=1,ndim
            do j=1,ndim
                idenspec(i,j) = idenspec(i,j) + &
                    & evecdag(i,k) * evec(k,j)
            end do
        end do
    end do

    print *, 'spectral␣representation␣of␣the␣unit␣matrix', &
        '␣''idenspec''␣as␣the␣sum␣of␣the␣outer␣products␣of', &
        '␣the␣eigenvectors␣of␣Hamiltonian␣operator␣''H'''
    do i= 1, ndim
        print '(40f10.4)', idenspec(1:ndim, i)
    end do

    do i = 1, ndim
        do j = 1, ndim
            re = realpart(idenspec(i,j)) - realpart(iden(i,j))
            im = imagpart(idenspec(i,j)) - imagpart(iden(i,j))
```

```fortran
            sum = abs(re) + abs(im)

            if ( sum > eps) then
                print *, '''idenspec''␣is␣not', &
                    '␣the␣identity␣matrix'
                go to 30
            end if
        end do
    end do
    30 if (i == ndim + 1) then
        print *, '''idenspec''␣is␣the␣identity␣matrix'
    end if

    print *, '------------------------'

    ! spectral representation 'Hspec' of the Hamiltonian
    ! operator matrix 'H'

    Hspec = matmul(spec, conjg(transpose(evec)))

    print *, 'spectral␣representation␣''Hspec''␣of␣the', &
        '␣Hamiltonian␣operator␣matrix␣''H'''
    do i= 1, ndim
        print '(40f10.4)', Hspec(1:ndim, i)
    end do

    do i = 1, ndim
        do j = 1, ndim
            re = realpart(Hspec(i,j)) - realpart(H(i,j))
            im = imagpart(Hspec(i,j)) - imagpart(H(i,j))
            sum = abs(re) + abs(im)

            if ( sum > eps) then
                print *, 'H␣/=␣matmul(spec,', &
                    '␣conjg(transpose(evec))'
                go to 300
            end if
        end do
    end do
    300 if (i == ndim + 1) then
```

```
        print *, 'H␣=␣matmul(spec,␣conjg(transpose(evec)))'
    end if

    print *, '------------------------'

    ! construct the function D(E) = |E     H| and show that the
    ! eigenvalues coincide with the zeros of the determinant
    do i = 1, ndim
        E = eval(i)*iden-H
        call zgbdi(E, ndim, ndim, ndim-1, ndim-1, ipiv, determinant)
        D(i) = determinant(1)
        print '(a2,␣f10.4,␣a14,␣2f11.4)', 'D(', eval(i), &
            ')␣=␣|E␣-␣H|␣=␣', determinant(1)
    end do

    do i = 1, ndim
        sum = abs(realpart(D(i)))+abs(imagpart(D(i)))
        if ( sum > eps) then
            print *, 'the␣eigenvalues␣do␣not␣coincide', &
                '␣with␣the␣zeros␣of␣the␣determinant'
            go to 90
        end if

    end do
    90 if (i == ndim + 1) then
        print *, 'the␣eigenvalues␣coincide␣with␣the␣zeros', &
        '␣of␣the␣determinant'
    end if

    print *, '------------------------'

end program hrmosc

function delta(x,y)
    ! function 'delta' outputs the result
    ! of the Dirac delta function of the inputs

    use numtype
    implicit none
    integer :: x, y, delta

    if (x == y) then
        delta = 1
    else
        delta = 0
    end if

end function delta

function det(E, H, ndim, eps)
    ! The function 'det' calculates the determinant
    ! of E - H by multiplying the diagonals of
    ! its LU factorization by -1**s, where s is
    ! the number of row intercganhes

    use numtype
    implicit none
    integer, external :: delta
    integer :: ndim, i, j, ipiv(ndim), info, s, ii
    real(dp) :: E, mag, eps, det, diag(ndim), diagi(ndim)
    complex(dp), dimension(ndim, ndim) :: H, mat, iden

    ! create identity matrix 'iden' for
    ! 'E' = eigenvalue * 'iden'
    do i = 1, ndim
        do j = 1, ndim
            iden(i, j) = delta(i, j)
        end do
    end do

    mat = E * iden - H

    ! for 'H' diagonal, skip zgetrf
    do i = 1, ndim
        do j = 1, ndim
            mag = abs(realpart(mat(i, j))) + &
                & abs(imagpart(mat(i, j)))

            if ( i /= j .and. mag > eps) then
                go to 100
```

```fortran
            end if
        end do
    end do
    if (i == ndim + 1) then
        go to 110
    end if

    ! for 'H' not diagonal
    100 call zgetrf(ndim, ndim, mat, ndim, ipiv, info)
        ! if(info /= 0) stop ' zgetrf info /= 0'
        ! this can be ignored since we will not solve
        ! linear equations with this -- thusly no dividing
        ! by 0

    do i = 1, ndim
        mag = abs(realpart(mat(i, i))) + &
                & abs(imagpart(mat(i, i)))
        if (mag < eps) then
            det = 0._dp
            return
        end if
    end do

    ! calculate magnitude of determinant 'D' of 'mat'
    110 det = 1._dp

        ! start by putting diagonal elements in order from
        ! least to greatest so that the numbers don't blow up
        ! as ndim gets arbitrarily large
    do i = 1, ndim
        diag(i) = realpart(mat(i, i))
    end do

    diagi = diag
    do i = 1, ndim
        ii = minloc(diagi,dim=1)
        diag(i) = diagi(ii)
        diagi(ii) = huge(0._dp)
    end do

    do i = 1, ndim
        det = det * diag(i)
    end do

    ! sign of determinant
    s = 0
    do i = 1, ndim
        if (ipiv(i) == i) then
            s = s + 0
        else
            s = s + 1
        end if
    end do

    det = (-1._dp)**s * det

end function det

subroutine zgbdi(abd, lda, n, ml, mu, ipvt, det)
    ! the LINPACK subroutine zgbdi outputs the
    ! complex determinant det(1) of a complex
    ! matrix abd

    use numtype
    implicit none
    complex(dp) :: abd(lda, n), det(2)
    integer :: lda, n, ml, mu, ipvt(n)

    integer :: i, m
    complex(dp) :: zdum, zdumr, zdumi
    real(dp) :: ten, cabs1, dreal, dimag

      dreal(zdumr) = zdumr
      dimag(zdumi) = (0.0d0,-1.0d0)*zdumi
      cabs1(zdum) = dabs(dreal(zdum)) + dabs(dimag(zdum))
      m = ml + mu + 1
      det(1) = (1.0d0,0.0d0)
      det(2) = (0.0d0,0.0d0)
      ten = 10.0d0
      do 50 i = 1, n
```

```
          if (ipvt(i) .ne. i) det(1) = -det(1)
          det(1) = abd(m,i)*det(1)
          exit
          if (cabs1(det(1)) .eq. 0.0d0) go to 60
   10     if (cabs1(det(1)) .ge. 1.0d0) go to 20
             det(1) = dcmplx(ten,0.0d0)*det(1)
             det(2) = det(2) - (1.0d0,0.0d0)
          go to 10
   20     continue
   30     if (cabs1(det(1)) .lt. ten) go to 40
             det(1) = det(1)/dcmplx(ten,0.0d0)
             det(2) = det(2) + (1.0d0,0.0d0)
          go to 30
   40     continue
   50  continue
   60  continue
       return

end subroutine
```

The full output of the code can be seen below with `ndim` = 4, `Mass` = `Omega` = 1 in Listing 4. These parameters were inputted to check if the code works out. The code does return the expected values for these inputs, so the check was successful.

Listing 4: Output of `Fortran 90` Code `hrmosc.f90` with `ndim` = 4, `Mass` = `Omega` = 1

```

------------------------
 lowering operator 'a' =
    0.0000     0.0000     1.0000     0.0000     0.0000     0.0000
0.0000     0.0000
    0.0000     0.0000     0.0000     0.0000     1.4142     0.0000
0.0000     0.0000
    0.0000     0.0000     0.0000     0.0000     0.0000     0.0000
1.7321     0.0000
    0.0000     0.0000     0.0000     0.0000     0.0000     0.0000
0.0000     0.0000
 ------------------------
 raising operator 'adag' =
    0.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000
0.0000   -0.0000
    1.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000
0.0000   -0.0000
    0.0000   -0.0000    1.4142   -0.0000    0.0000   -0.0000
0.0000   -0.0000
    0.0000   -0.0000    0.0000   -0.0000    1.7321   -0.0000
0.0000   -0.0000
 -------------------------
 position operator 'X' =
    0.0000    0.0000    0.7071    0.0000    0.0000    0.0000
0.0000    0.0000
    0.7071    0.0000    0.0000    0.0000    1.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000    0.0000    0.0000
1.2247    0.0000
    0.0000    0.0000    0.0000    0.0000    1.2247    0.0000
0.0000    0.0000
 -------------------------
 momentum operator 'P' =
    0.0000    0.0000    0.0000   -0.7071    0.0000    0.0000
0.0000    0.0000
    0.0000    0.7071    0.0000    0.0000    0.0000   -1.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    1.0000    0.0000    0.0000
0.0000   -1.2247
    0.0000    0.0000    0.0000    0.0000    0.0000    1.2247
0.0000    0.0000
 -------------------------
 Hamiltonian operator 'H' =
    0.5000    0.0000    0.0000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    1.5000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    2.5000    0.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
1.5000    0.0000
 -------------------------
 eigenvalue array 'eval' =
    0.5000
    1.5000
    1.5000
    2.5000
 ------------------------
 the eigenvectors are the columns of 'evec' =
    1.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
1.0000    0.0000
   -0.0000   -0.0000    1.0000    0.0000    0.0000    0.0000
0.0000    0.0000
   -0.0000   -0.0000   -0.0000   -0.0000    1.0000    0.0000
0.0000    0.0000
 evec(1:ndim,i) is the corresponding eigenvector to the eigenvalue eval(i
 ------------------------
 the eigenvectors are orthogonal if the following is the identity matrix:
    1.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    1.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
1.0000    0.0000
 the eigenvectors are orthogonal
 ------------------------
 spectral representation of the unit matrix 'idenspec' as the sum of the
    1.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    1.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    0.0000    0.0000
1.0000    0.0000
 'idenspec' is the identity matrix
 ------------------------
 spectral representation 'Hspec' of the Hamiltonian operator matrix 'H'
    0.5000    0.0000    0.0000    0.0000    0.0000    0.0000
```

```
0.0000    0.0000
     0.0000     0.0000     1.5000     0.0000     0.0000     0.0000
0.0000    0.0000
     0.0000     0.0000     0.0000     0.0000     2.5000     0.0000
0.0000    0.0000
     0.0000     0.0000     0.0000     0.0000     0.0000     0.0000
1.5000    0.0000
 H = matmul(spec, conjg(transpose(evec)))
 -------------------------
D(     0.5000) = |E - H| =       0.0000    -0.0000
D(     1.5000) = |E - H| =       0.0000    -0.0000
D(     1.5000) = |E - H| =       0.0000    -0.0000
D(     2.5000) = |E - H| =       0.0000    -0.0000
 the eigenvalues coincide with the zeros of the determinant
 -------------------------
```

The output of the code for `ndim` $= 4$ and `Mass` $= 2$, and `Omega` $= \sqrt{2}$, as given in the problem, is shown below in Listing 5. The choice of `ndim` $= 4$ was kept for this output so that the answers are easier to see.

Listing 5: Output of `Fortran 90` Code `hrmosc.f90` with `ndim` $= 4$, `Mass` $= 2$, and `Omega` $= \sqrt{2}$

```

 -------------------------
 lowering operator 'a' =
     0.0000     0.0000     1.0000     0.0000     0.0000     0.0000
0.0000    0.0000
     0.0000     0.0000     0.0000     0.0000     1.4142     0.0000
0.0000    0.0000
     0.0000     0.0000     0.0000     0.0000     0.0000     0.0000
1.7321    0.0000
     0.0000     0.0000     0.0000     0.0000     0.0000     0.0000
0.0000    0.0000
 -------------------------
 raising operator 'adag' =
     0.0000    -0.0000     0.0000    -0.0000     0.0000    -0.0000
0.0000   -0.0000
     1.0000    -0.0000     0.0000    -0.0000     0.0000    -0.0000
0.0000   -0.0000
     0.0000    -0.0000     1.4142    -0.0000     0.0000    -0.0000
```

```
0.0000   -0.0000
    0.0000   -0.0000    0.0000   -0.0000    1.7321   -0.0000
0.0000   -0.0000
 ------------------------
 position operator 'X' =
    0.0000    0.0000    0.7071    0.0000    0.0000    0.0000
0.0000    0.0000
    0.7071    0.0000    0.0000    0.0000    1.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000    0.0000    0.0000
1.2247    0.0000
    0.0000    0.0000    0.0000    0.0000    1.2247    0.0000
0.0000    0.0000
 ------------------------
 momentum operator 'P' =
    0.0000    0.0000    0.0000   -0.7071    0.0000    0.0000
0.0000    0.0000
    0.0000    0.7071    0.0000    0.0000    0.0000   -1.0000
0.0000    0.0000
    0.0000    0.0000    0.0000    1.0000    0.0000    0.0000
0.0000   -1.2247
    0.0000    0.0000    0.0000    0.0000    0.0000    1.2247
0.0000    0.0000
 ------------------------
 Hamiltonian operator 'H' =
    1.1250    0.0000    0.0000    0.0000    1.2374    0.0000
0.0000    0.0000
    0.0000    0.0000    3.3750    0.0000    0.0000    0.0000
2.1433    0.0000
    1.2374    0.0000    0.0000    0.0000    5.6250    0.0000
0.0000    0.0000
    0.0000    0.0000    2.1433    0.0000    0.0000    0.0000
3.3750    0.0000
 ------------------------
 eigenvalue array 'eval' =
    0.8072
    1.2317
    5.5183
    5.9428
 ------------------------
 the eigenvectors are the columns of 'evec' =
    0.9686    0.0000   -0.0000    0.0000   -0.2488    0.0000
0.0000    0.0000
    0.0000   -0.0000    0.7071   -0.0000    0.0000   -0.0000
-0.7071   -0.0000
    0.0000    0.0000    0.7071    0.0000    0.0000    0.0000
0.7071    0.0000
   -0.2488    0.0000    0.0000    0.0000   -0.9686    0.0000
0.0000    0.0000
 evec(1:ndim,i) is the corresponding eigenvector to the eigenvalue eval(i
 ------------------------
 the eigenvectors are orthogonal if the following is the identity matrix:
    1.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000    0.0000    0.0000
-0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    1.0000    0.0000
-0.0000    0.0000
    0.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000
1.0000    0.0000
 the eigenvectors are orthogonal
 ------------------------
 spectral representation of the unit matrix 'idenspec' as the sum of the
    1.0000    0.0000    0.0000    0.0000    0.0000    0.0000
0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000    0.0000    0.0000
-0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000    1.0000    0.0000
-0.0000    0.0000
    0.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000
1.0000    0.0000
 'idenspec' is the identity matrix
 ------------------------
 spectral representation 'Hspec' of the Hamiltonian operator matrix 'H'
    1.1250    0.0000    0.0000    0.0000    1.2374    0.0000
-0.0000    0.0000
    0.0000    0.0000    3.3750    0.0000   -0.0000    0.0000
2.1433    0.0000
    1.2374    0.0000   -0.0000    0.0000    5.6250    0.0000
0.0000    0.0000
```

```
   -0.0000     0.0000     2.1433     0.0000    -0.0000     0.0000
3.3750    0.0000
 H = matmul(spec, conjg(transpose(evec)))
 ------------------------
D(     0.8072) = |E - H| =       0.0000    -0.0000
D(     1.2317) = |E - H| =       0.0000    -0.0000
D(     5.5183) = |E - H| =       0.0000    -0.0000
D(     5.9428) = |E - H| =       0.0000    -0.0000
 the eigenvalues coincide with the zeros of the determinant
 ------------------------
```

Since the matrices printed in the output are hard to read for `ndim` = 20, which was given in the problem, printing that will be left to the reader when running the code. We included the output of the important parts below in Listing 6 that do not require diving through large matrices.

Listing 6: Selected parts of output of `Fortran 90` Code `hrmosc.f90` for `ndim` = 20

```

------------------------
 eigenvalue array 'eval' =
    0.7071
    2.1212
    3.5362
    4.9293
    6.3942
    7.1696
    9.1058
    9.5202
   12.8683
   13.4361
   17.9818
   18.5311
   24.5263
   25.0557
   32.8431
   33.3589
   43.6129
   44.1192
   58.5919
```

```
  59.0913
------------------------
evec(1:ndim,i) is the corresponding eigenvector to the eigenvalue eval(i
------------------------
the eigenvectors are orthogonal
------------------------
'idenspec' is the identity matrix
------------------------
H = matmul(spec, conjg(transpose(evec)))
------------------------
D(     0.7071) = |E - H| =      0.0000    -0.0000
D(     2.1212) = |E - H| =      0.0000    -0.0000
D(     3.5362) = |E - H| =      0.0000    -0.0000
D(     4.9293) = |E - H| =      0.0000    -0.0000
D(     6.3942) = |E - H| =      0.0000    -0.0000
D(     7.1696) = |E - H| =      0.0000    -0.0000
D(     9.1058) = |E - H| =      0.0000    -0.0000
D(     9.5202) = |E - H| =      0.0000    -0.0000
D(    12.8683) = |E - H| =      0.0000    -0.0000
D(    13.4361) = |E - H| =      0.0000    -0.0000
D(    17.9818) = |E - H| =      0.0000    -0.0000
D(    18.5311) = |E - H| =      0.0000    -0.0000
D(    24.5263) = |E - H| =      0.0000    -0.0000
D(    25.0557) = |E - H| =      0.0000    -0.0000
D(    32.8431) = |E - H| =      0.0000    -0.0000
D(    33.3589) = |E - H| =      0.0000    -0.0000
D(    43.6129) = |E - H| =      0.0000    -0.0000
D(    44.1192) = |E - H| =      0.0000    -0.0000
D(    58.5919) = |E - H| =      0.0000    -0.0000
D(    59.0913) = |E - H| =      0.0000    -0.0000
the eigenvalues coincide with the zeros of the determinant
------------------------
```

3 Problem 2

The `Makefile` used for problem 2 can be found below in Listing 7. It gives instructions for the `gfortran` compiler on how to compile the `Fortran 90` modules `numtype.f90` and `cheby.f90` and code `zeros.f90` into object files and link them with the library `Accelerate`, which includes the linear algebra package `LAPACK`, in an executable `zer`. The code can be compiled by typing `make` into the terminal when in the directory `~/src` and executed by typing `zer`. Flags have been added to the compiling instructions for optimization. By typing `make clean` into the terminal the executable, object, `*.mod`, and `fort.*` files will be removed from the directory leaving only the Makefile and .f90 files required to run the code again from scratch.

Listing 7: `Makefile`

```

objs1 = numtype.o cheby.o zeros.o

prog1 = zer

f90 = gfortran

f90flags = -O3 -funroll-loops -ftree-vectorize -fexternal-blas

libs = -framework Accelerate

ldflags = $(libs)

all: $(prog1)

$(prog1): $(objs1)
	$(f90) $(ldflags) -o $@ $(objs1)

clean:
	rm -f $(prog1) *.{o,mod} fort.*

.suffixes: $(suffixes) .f90

%.o: %.f90
	$(f90) $(f90flags) -c $<
```

The module `numtype` can be found below in Listing 8. Included in this module are the desired precision of real numbers `dp`, the value of the constant π, numbers that can be multiplied by real data types to turn them into complex data types with precision `dp` as needed, and a real parameter `tiny`.

Listing 8: `Fortran 90` Module `numtype.f90`

```

module numtype
    save
    integer, parameter :: dp = selected_real_kind(15,307)
    !integer, parameter :: qp = selected_real_kind(33,4931)
    real(dp), parameter :: pi = 4*atan(1._dp)
    !defining a complex number
    complex(dp), parameter :: z0 = (0._dp, 0._dp), z1 = (1._dp, 0._dp), &
        zi = (0._dp,1._dp)
    real(dp), parameter :: tiny = 1.e-30_dp

end module numtype
```

The module `cheby.f90` can be found below in Listing 9. This module details the subroutines involved in using the chebyshev polynomials to approximate functions.

Listing 9: Module `cheby.f90`

```

module chebyshev

    use numtype
    implicit none
    integer, parameter :: maxch = 50
    real(dp), dimension(0:maxch) :: cheb, chder, chder2
    real(dp), dimension(maxch) :: z0
    integer :: iz0

    contains

        subroutine chebyex(func,n,a,ya,yb)
        !   func([ya,yb]) = sum_{i=0}^n  a_i T_i

            real(dp), external :: func
            integer :: n
            real(dp), dimension(0:maxch) :: f, a
            real(dp) :: ya, yb, aa, bb, x, ss
            integer :: i, j

```

```fortran
        if ( n > maxch ) stop '␣␣n␣>␣maxch␣'
        aa = (yb-ya)/2; bb = (yb+ya)/2
        do i = 0, n
            x = cos(pi/(n+1)*(i+0.5_dp))
            f(i) = func(aa*x+bb)
        end do
        do j = 0, n
            ss = 0._dp
            do i = 0, n
                ss = ss + &
                    f(i)*cos((pi/(n+1))*j*(i+0.5_dp))
            end do
            a(j) = 2._dp*ss/(n+1)
        end do
        a(0) = 0.5_dp*a(0)

    end subroutine chebyex

    subroutine chebyderiv(a,n,der,ya,yb) !

        integer :: n
        real(dp) :: ya, yb, a(0:maxch), der(0:maxch)
        integer :: j

        der(n) = 0._dp; der(n-1) = 2*n*a(n)
        do j = n-1, 1, -1
            der(j-1) = der(j+1)+2*j*a(j)
        end do
        der(0) = der(0)/2
        der(0:n-1) = der(0:n-1)*2/(yb-ya)

    end subroutine chebyderiv

    function cheby(y,a,n,ya,yb) result(t)
    ! func(y) =  sum_{i=0}^n  a_i T_i (x)

        implicit none
        integer :: n
        real(dp) :: y, ya, yb
        real(dp) :: a(0:maxch)
        real(dp) :: aa, bb, x, t, y0, y1
        integer :: k

        aa = (yb-ya)/2; bb = (yb+ya)/2
        x = (y-bb)/aa
        y1 = 0._dp; y0 = a(n)
        do k = n-1, 0, -1
            t = y1; y1 = y0
            y0 = a(k)+2*x*y1-t
        end do
        t = y0-x*y1

    end function cheby

    subroutine chebyzero(n,a,ya,yb,z0,iz0)
    !   find zero by using Boyd's method

        integer :: n, iz0
        real(dp), dimension(0:maxch) :: a
        integer :: j
        real(dp), dimension(maxch) :: wr0, wi0, z0, wwr0
        real(dp) :: ya, yb

        call boyd(n,a,wr0,wi0)
        wwr0(1:n) = wr0(1:n)*(yb-ya)/2+(yb+ya)/2

        iz0 = 0
        do j = 1, n
            if( wi0(j) == 0._dp .and. &
              -1 <= wr0(j) .and. wr0(j) <= 1 ) then
                    iz0 = iz0+1;  z0(iz0) = wwr0(j)
            end if
        end do

        contains

            subroutine boyd(n,a,wr,wi)

                integer :: n, j, ie
                real(dp) :: a(0:maxch)
                real(dp) :: wr(maxch), wi(maxch)
                integer, parameter :: lwork=4*maxch
                real(dp) :: aamat(maxch,maxch),  &
                    work(lwork), rwork(lwork), &
                    vl(1), vr(1)

                if (abs(a(n)) == 0._dp) stop 'a(n)=0'
                aamat(1:n,1:n) = 0._dp
                aamat(1,2) = 1._dp
                do j = 2, n-1
                    aamat(j,j-1) = 0.5_dp
                    aamat(j,j+1) = 0.5_dp
                end do
                aamat(n,1:n) = -a(0:n-1)/(2*a(n))
                aamat(n,n-1) = aamat(n,n-1) + 0.5_dp

                ie = 0
                call dgeev('n','n',n,aamat,maxch,wr,&
                    wi,vl,1,vr,1,work,lwork,rwork,ie)
                if( ie /= 0 ) stop '␣boyd:␣ie␣/=␣0␣'

            end subroutine boyd

    end subroutine chebyzero

    subroutine root_polish(func,zz,dz,eps,maxf)

        real(dp), external :: func
        real(dp) :: zz, dz, eps, z1, z2, z3, &
            f1, f2, f3, a12, a23, a31
        integer :: i, maxf

        z1 = zz+dz;   f1 = func(z1)
        z2 = zz-dz;   f2 = func(z2)
        z3 = zz;      f3 = func(z3)

        do i = 1,maxf
            a23 = (z2-z3)*f2*f3
            a31 = (z3-z1)*f1*f3
            a12 = (z1-z2)*f1*f2
```

```
                zz = (z1*a23+z2*a31+z3*a12)/(a23+a31+a12)
                if ( abs(zz-z3) < eps ) exit
                z1 = z2;  f1 = f2
                z2 = z3;  f2 = f3
                z3 = zz;  f3 = func(z3)
            end do

        end subroutine root_polish

end module chebyshev
```

The code `zeros.f90` can be found below in Listing 10. It begins with the module `setup`, which defines the parameters. The Newton-Raphson method is multivariable, but we can just set the initial parameters to 1 in this case because we are only looking at a one-dimensional function. The program has a parameter `eps` at the beginning which can be changed as well depending on the situation. The code defines the given function externally as `func`, and the method is used iteratively at each step with the `LAPACK` function `dgesv` to solve the system of linear equations described in the introduction. To find the poles, the same method is repeated but for the inverse of the function, `funcinv`. The residues for this function were found as the first term of the Laurent series expansion around each of the poles and happened to follow the pattern defined in the code. Then the Newton-Raphson method results are compared to the Chebyshev results. The Newton-Raphson results seem to be slightly more accurate with the given parameters (function values are closer to zero), and the Chebyshev residues become less accurate as the poles get farther from `xx` = 1.

Listing 10: `Fortran 90` Code `zeros.f90`

```

module setup

    use numtype
    implicit none
    integer, parameter :: nv = 1, lda = 1, nrhs = 1
    real(dp) :: x(lda), f(lda), deriv(lda, lda)

end module setup

program zeros
```

```

    use setup
    use chebyshev
    implicit none

    ! Newton-Raphson
    real(dp) :: dx(lda, nrhs), ff, res
    integer :: info, i, ipiv(lda), maxstep, j
    real(dp), parameter :: eps = 1.e-15_dp

    ! Chebyshev
    integer :: n, np, maxf
    real(dp) :: ya, yb, dxx, xx, ress, xxx(maxch)
    real(dp), external :: funcc, funccinv


    print *, '------------------'

    ! use Newton-Raphson to locate the zeros of 'func' for x \in [0, 5]
    print *, 'zeros␣for␣x␣\in␣[0,␣5]:'

    do j = 0, 5

        x(1:nv) = (/ j - 0.1_dp /)

        maxstep = 50
        print '((9x,a),4x,2(9x,a))','x', 'f_1', '|f|'

        do i = 1, maxstep

            call func(ff)

            ! print '(f10.4,4x,2e12.3)',x(1:nv),f(1:nv),ff
            if (ff <= eps) exit

            dx(1:nv,1) = -f(1:nv)

            info = 0
            call dgesv(nv, nrhs, deriv, lda, ipiv, dx, lda, info)
            if (info /= 0) stop 'info␣/=␣0'

            x(1:nv) = x(1:nv) + dx(1:nv,1)

        end do

        print '(f10.4,4x,2e12.3)',x(1:nv),f(1:nv),ff

    end do

    print *, '------------------'

    ! locate the poles of 'func' for x \in [0, 5]
    print *, 'poles␣for␣x␣\in␣[0,␣5]'

    do j = 1, 5

        x(1:nv) = (/ j + 0.1_dp /)

        maxstep = 9492
        print '((9x,a),4x,(9x,a))','x', 'residue'

        do i = 1, maxstep

            call funcinv(ff)

            !print '(f10.4,4x,2e12.3)',x(1:nv),f(1:nv),ff
            if (ff <= eps) exit

            dx(1:nv,1) = -f(1:nv)

            info = 0
            call dgesv(nv, nrhs, deriv, lda, ipiv, dx, lda, info)
            if (info /= 0) stop 'info␣/=␣0'

            x(1:nv) = x(1:nv) + dx(1:nv,1)

        end do

        ! residue
        res = pi * j**2

        print '(f10.4,8x,f12.3)',x(1:nv), res

    end do

    print *, '------------------'

    ! use Chebyshev to locate the zeros of 'funcc' for xx \in [0, 5]

    ya = 0
    yb = 5

    n = 9
    np = 50
    dxx = ( yb - ya ) / np

    call chebyex( funcc, n, cheb, ya, yb )

    call chebyzero( n, cheb, ya, yb, z0, iz0 )

    maxf = 10

    do i = 1, iz0

        xx = z0(i)
        call root_polish( funcc, xx, dxx, eps, maxf )
        if (xx < eps .or. xx > yb) cycle
        print *, "f(x)␣=␣0", i, xx, funcc(xx)

    end do

    print *, '------------------'

    ! use Chebyshev to locate the poles of 'func' for x \in [0, 5]

    ya = 0
    yb = 5

    n = 9
    np = 50
```

```fortran
    dxx = ( yb - ya ) / np

    call chebyex( funccinv, n, cheb, ya, yb )

    call chebyzero( n, cheb, ya, yb, z0, iz0 )

    maxf = 10

    j = 0

    do i = 1, iz0

        xx = z0(i)
        call root_polish( funccinv, xx, dxx, eps, maxf )

        xxx(i) = xx
        if (i > 1) then
            if (abs(xx - xxx(i-1)) < eps) then
                j = j + 1
            cycle
            end if
        end if

        if (xx < eps .or. xx > yb) cycle

        ! residua
        call chebyex( funccinv, 1, cheb, xx-eps, xx+eps )

        ress = eps / cheb(1)

        print *, "1/f(x) = 0", i-j, xx, funccinv(xx), ress

    end do

    print *, '------------------'

end program zeros

! Newton-Raphson
subroutine func(ff)
```

```fortran

    use setup
    implicit none

    real(dp) :: ff

    f(1) = (x(1)*pi)**2 / tan( pi * x(1) )
    ff = norm2(f(1:nv))

    deriv(1, 1) = pi**2 * x(1) * (2 / tan( pi * x(1)) - pi * x(1) / sin(

end subroutine func

subroutine funcinv(ff)

    use setup
    implicit none

    real(dp) :: ff

    f(1) = tan( pi * x(1) ) / (x(1) * pi)**2
    ff = norm2(f(1:nv))

    deriv(1, 1) = pi * x(1) / cos( pi * x(1) )**2 - 2 * tan( pi * x(1) )

end subroutine funcinv

! Chebyshev
function funcc(x) result(y)

    use numtype
    implicit none
    real(dp) :: x, y

    y = (x * pi)**2 / tan(pi * x)

end function funcc

function funccinv(x) result(y)

```

```
    use numtype
    implicit none
    real(dp) :: x, y

    y = 1 / ( (x * pi)**2 / tan(pi * x) )

end function funccinv
```

The output of running the executable `zer` can be found below in Listing 11. The Newton-Raphson results are labeled, and the Chebyshev results by column list the number of the zero, the x value, the function value, and for the last part the residua.

Listing 11: Output of `Fortran 90` Code `zeros.f90`

```

 ------------------
 zeros for x \in [0, 5]:
         x             f_1           |f|
    0.0000       0.396E-15   0.396E-15
         x             f_1           |f|
    0.5000       0.151E-15   0.151E-15
         x             f_1           |f|
    1.5000       0.408E-14   0.408E-14
         x             f_1           |f|
    2.5000       0.189E-13   0.189E-13
         x             f_1           |f|
    3.5000       0.518E-13   0.518E-13
         x             f_1           |f|
    4.5000       0.110E-12   0.110E-12
 ------------------
 poles for x \in [0, 5]
         x             residue
    1.0000               3.142
         x             residue
    2.0000              12.566
         x             residue
    3.0000              28.274
         x             residue
    4.0000              50.265
         x             residue
```

```
   5.0000              78.540
 ------------------
 f(x) = 0          1   4.5000000000000000        1.1014077763464995E-013
 f(x) = 0          2   3.5000000000000000        5.1822066843189195E-014
 f(x) = 0          3   2.5000000000000004       -9.0689027627429528E-014
 f(x) = 0          4   1.5000000000000000        4.0792880605425908E-015
 f(x) = 0          5  0.50000000000000000        1.5108474298305889E-016
 ------------------
 1/f(x) = 0          1   1.0000000000000002         7.7583022446256687E-0
3.1430010208962971
 1/f(x) = 0          2   2.0000000000000000        -6.2041331616705060E-0
12.572004083585185
 1/f(x) = 0          3   3.0000000000000000        -4.1360887744470032E-0
35.358761485083321
 1/f(x) = 0          4   4.0000000000000000        -3.1020665808352530E-0
41.906680278617280
 1/f(x) = 0          5   5.0000000000000000        -2.4816532646682023E-0
49.109390951504615
 ------------------
```

4 Summary and conclusions

Ultimately, this midterm showed us the power of different standard subroutines in `Fortran 90` when applied to computational physics. The subroutines in the package `LAPACK` are invaluable in numerical approximations to quantum mechanics because of its matrix formulation since the time of Heisenberg. There is also a lot of value in the code for being able to find the zeros of functions, for applications such as mentioned before finding the equilibrium points of a system. We were surprised to see that the Newton-Raphson method returned function values closer to 0 than the Chebyshev method did within the parameters we used. All in all, `Fortran 90` codes like these have opened up a new era of physics that we are fortunate to be on the brink of.

References

[1] Wikipedia contributors. (2020, April 14). Mathematical formulation of quantum mechanics. In *Wikipedia, The Free Encyclopedia*. Retrieved 19:20, April 17, 2020, from `https://en.wikipedia.org/w/index.php?title=Mathematical_formulation_of_quantum_mechanics&oldid=950905216`

[2] Papp, Z. (2020). *Mastering Computational Physics* [(PHYS - 562) Lecture Notes]. Department of Physics, California State University, Long Beach.

[3] Wikipedia contributors. (2020, March 29). Quantum harmonic oscillator. In *Wikipedia, The Free Encyclopedia*. Retrieved 18:13, April 17, 2020, from `https://en.wikipedia.org/w/index.php?title=Quantum_harmonic_oscillator&oldid=947984310`

[4] Wikipedia contributors. (2020, April 17). Spectral theorem. In *Wikipedia, The Free Encyclopedia*. Retrieved 21:51, April 19, 2020, from `https://en.wikipedia.org/w/index.php?title=Spectral_theorem&oldid=951531749`