Time-dependent Schrodinger Equation

Alberto Chimenti

Exercise n.7
Information Theory and Computation - 2019

Abstract

In the following is explained the chosen computational procedure to compute the solution of the Schrodinger equation with a time-dependent harmonic potential.

1 Theory

The considered Hamiltonian can be written as:

$$\mathcal{H} = \hat{p}^2 + (\hat{x} + x_0(t))^2 = \hat{T} + \hat{V}(t)$$

where $x_0(t) = t/T_{max}$ and $t \in [0 : T_{max}]$.
Here we are interested in computing the time evolution of the eigenfunctions. By using the unitary time evolution operator:

$$|\psi_x(t + \Delta t) >= \mathcal{U}|\psi_x(t) >= e^{-i\frac{\mathcal{H}\Delta t}{\hbar}}|\psi_x(t) >$$

In order to conserve the modulus squared of the wavefunction and exploit the form of the Hamiltonian, one can rewrite it, using the split operator, as:

$$|\psi_x(t + \Delta t) > \simeq e^{-i\frac{V\Delta t}{2\hbar}} e^{-i\frac{T\Delta t}{\hbar}} e^{-i\frac{V\Delta t}{2\hbar}} |\psi_x(t) >$$

Here we are neglecting the terms that come from the commutator, hence we are approximating it with a discrepancy of the order $O(\Delta t^3)$. This way one can compute the wavefunction in the next timestep in pieces using the fact that the position operator is already diagonal in the position space while the free particle term is diagonal in the momentum space.

Therefore it is possible to calculate, from right to left:

- compute the first product
- perform a discrete Fourier transform over the solution and compute the second product in the momentum space (where T is diagonal)
- perform a discrete inverse Fourier transformation on the solution of the second product and finally compute the last product in position space

For simplicity, all the physical constants have been set equal to 1.
One has to note that in order to computationally solve the problem, the space has to be discretized in both the position and momentum space. Indeed the discretization over x was formalized as:

$$V_i(t) = \left(x_i - \frac{t}{T_{max}}\right)^2 \qquad with \quad i \in [1 : N_{points}]$$

While the corresponding discretization over the momentum space has to be chosen equal to:

$$T_k = \begin{cases} \left(\frac{2\pi k}{N_{points}\Delta x}\right)^2 & \text{for } k \in [0 : \frac{N_{points}}{2}] \\ \left(\frac{2\pi(N_{points}-k)}{N_{points}\Delta x}\right)^2 & \text{for } k \in [\frac{N_{points}}{2}+1 : N_{points}] \end{cases}$$

This is done in order to respect the normalization and the periodicity of the discrete Fourier transform.

2 Code Development

2.1 Fourier Transform

The discrete Fourier transform has been computed using the library **FFTW3**. In particular it has the following syntax:

```
[...]
    type(C_PTR) :: forward, backward
    integer FFTW_FORWARD,FFTW_BACKWARD
    parameter (FFTW_FORWARD=-1,FFTW_BACKWARD=1)
    integer FFTW_ESTIMATE,FFTW_MEASURE
    parameter (FFTW_ESTIMATE=0,FFTW_MEASURE=1)
[...]
```

This part defines the needed parameters. The **C__PTR** type is the suggested format of the plan variables, it is inherited from the intrinsic fortran module **iso__c__binding** and has the structure of a C pointer (INTEGER*8).

```
[...]
    ! initialize plans for ffts
    call dfftw_plan_dft_1d(forward, grid%N, temp, temp, FFTW_FORWARD, FFTW_ESTIMATE)
    call dfftw_plan_dft_1d(backward, grid%N, temp, temp, FFTW_BACKWARD, FFTW_ESTIMATE)
[...]
        ! fft
        call dfftw_execute_dft(forward, temp, temp)
[...]
        ! back fft
        call dfftw_execute_dft(backward, temp, temp)
[...]
    call dfftw_destroy_plan(forward)
    call dfftw_destroy_plan(backward)
[...]
```

Here we create the plans for forward and backward transform and then destroy them after performing the transforms. It is important to underline here that since we are performing a discrete Fourier transform, the result of the forward-backward transformation has to be rescaled by dividing by the size of the grid vector.

2.2 Gridding and kinetic term

In the following are reported the code chunks used to compute the spacing vectors.

```
    TYPE grid1d
        double precision :: min
        double precision :: max
        integer*4 :: N
        double precision :: h
    END TYPE grid1d

    contains

```

```fortran
    function init_grid1d(x_min, x_max, N) result(grid)
        ! Initializes the grid type object
        type(grid1d) :: grid
        double precision :: x_min
        double precision :: x_max
        integer :: N

        grid%min = x_min
        grid%max = x_max
        grid%N = N
        grid%h = (x_max - x_min)/N
    end function

    [...]

    function momentum_grid(grid) result(p)
        type(grid1d) :: grid
        double precision, dimension(grid%N) :: p
        integer :: jj

        do jj=1, grid%N/2
            p(jj) = (2*pi*jj/(grid%N*grid%h))
        end do
        do jj=1+grid%N/2, grid%N
            p(jj) = (2*pi*(grid%N - jj)/(grid%N*grid%h))
        end do
    end function


    function timeprop(psi0, grid, tgrid, hbar, m, omega) result(psi)
        [...]
        k = cmplx(0., -tgrid%h*0.5/hbar)
        do ii=2, tgrid%N
            [...]
            ! fft
            call dfftw_execute_dft(forward, temp, temp)

            ! kinetic term
            temp = temp * exp(k*(p_grid**2)/m)

            ! back fft
            call dfftw_execute_dft(backward, temp, temp)
            [...]
        end do
        [...]
```

First of all a **grid1d** object was defined, in order to keep the discretization variables tidy. Then a dedicated function was written to compute the momentum vector in the discretized momentum space and finally used in the function **timeprop**.

2.3 Parameters, Groundstate and normalization

Since the computation of the initial ground state was out of our scope, in order to reduce the computational cost, a dedicated function was written to compute the discretized version from its analytical expression.

We recall that for simplicity all the physical constants have been set equal to 1 too in order to keep it coherent with the rest of the calculations. The following variables have been fixed at the corresponding values: $t_{min} = 0$; $x_{max} = 5 = -x_{min}$; $N_{points} = 5000$.

The code was run for different instantiations of T_{max} and N_{steps}.

The resulting wavefunctions have been integrated over the considered region in order to check their normalization. This has been done using Simpson's method with the **integrate.simps** function contained in the **scipy** library for Python.

3 Results

In the following we plot the most interesting results of the computation obtained by keeping fixed $N_{steps} = 5000$ and changing the value of T_{max}. It is suggested to take the first and the last labelled wavefunctions as references to interpret the following plots. The reader is invited to take a look at the corrsponding .gif animated plots in the correspondent folders.

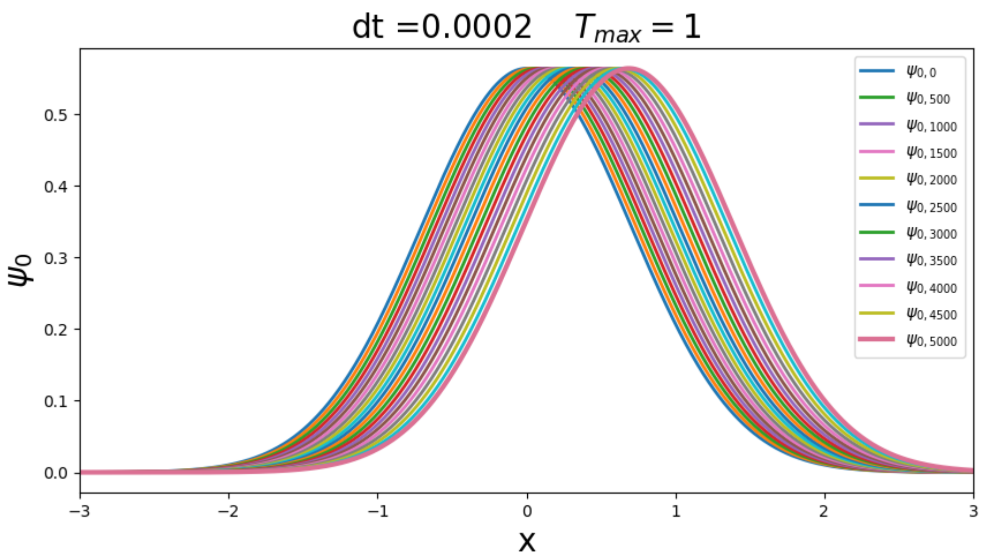

Figure 1: Groundstate wavefunction (only 20 plotted among the 5000 computed with steps=$\frac{N_{steps}}{20}$).
Name: **evolution_short.gif**

Here we can see that the 20 plotted curves are evenly spaced between each other, which is what we should expect from choosing $T_{max} = 1$ equal to the actual last point of the potential translation. However here one can see that its peak doesn't reach $x = 1$. Therefore we can deduce that it is probably not moving exactly with the minimum of the potential.

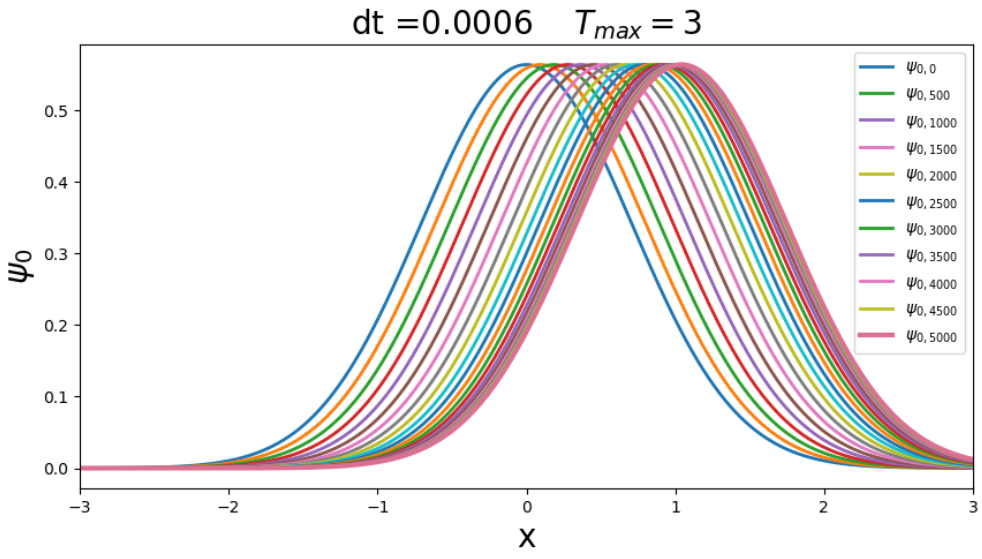

Figure 2: Groundstate wavefunction (only 20 plotted among the 5000 computed with steps=$\frac{N_{steps}}{20}$).
Name: **evolution_long.gif**

The interesting thing here is that we don't see even spacing between the curves anymore, even if the number of steps didn't vary. The particle here behaves in a way which might let one think of some sort of inertia acting on it. It is indeed decellerating the closer it gets to the final value, finally oscillationg around it.

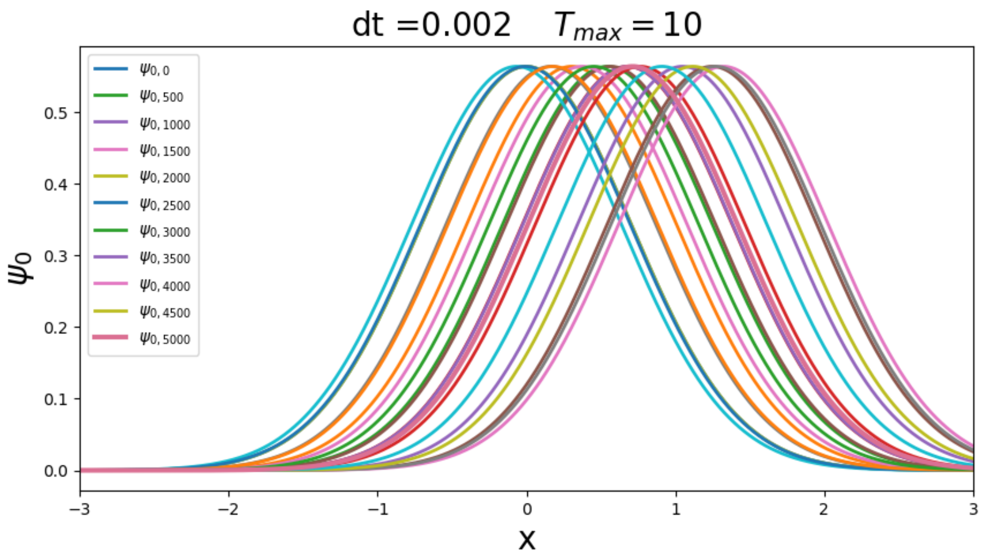

Figure 3: Groundstate wavefunction (only 20 plotted among the 5000 computed with steps=$\frac{N_{steps}}{20}$).
Name: **evolution_out.gif**

Here we can see that either the timestep is too high and the wavefunction is bouncing from left to right slowly reaching $x = 1$ (see .gif animated plot).

4 Self-evaluation

As a first improvement one could introduce the physical constants and play with them to explore physical or unphysical behaviour of the algorithm. Moreover it could be interesting to enlarge the spatial cutoffs and remove normalization in the time dependent term in the potential. The latter should probably have some effects on the inertial behaviour previously seen which is most probably not physical.