Time-dependent Schrödinger equation

Abstract

In this work we write a Fortran program to solve the time-dependent Schrödinger equation in the case of shifted quadratic potential.

1 Theory

Let us consider the following Hamiltonian:

$$H = \frac{\hat{p}^2}{2} + \frac{(\hat{x} - x_0(t))^2}{2}, \tag{1}$$

where $\hat{x}$ is the position operator, $\hat{p} = -i\hbar\frac{\partial}{\partial q}$ is the momentum operator, $t \in [0, T]$ is the time and $q_0(t) = t/T$, $T > 0$, is the velocity of the potential. Essentially, we are dealing with a moving potential.

In order to solve the corresponding time-dependent Schrödinger equation

$$i\frac{\partial}{\partial t}\psi(x,t) = H(t)\psi(x,t) \qquad \Rightarrow \qquad |\psi(t)\rangle = \mathcal{U}\,|\psi(t)\rangle\,, \quad \mathcal{U} = e^{-\frac{iHt}{\hbar}}$$

we use the split-operator method. The starting point is that, given $H = T + V$, T is diagonal in the momentum eigenbasis, whereas V is diagonal in the position eigenbasis. But we know that these two basis are connected by a Fourier transform, so we can exploit this property. Let us consider the following expansion:

$$e^{-iH\Delta t} = e^{-i(T+V)\Delta t} \approx \exp\left(-\frac{iV\Delta t}{2}\right) e^{-iT\Delta t} \exp\left(-\frac{iV\Delta t}{2}\right) + \mathcal{O}(\Delta t^3). \tag{2}$$

Notice that this is an approximation, since in general T and V do not commute. Now, the action of the two operators is splitted and we can apply them independently:

$$\begin{cases} \exp\left(-\frac{iV\Delta t}{2}\right) |\psi(x)\rangle = \exp\left(-i\frac{V(x_i)\Delta t}{2}\right)\psi(x_i) \\ e^{-iT\Delta t}\,|\psi(p)\rangle = \exp\left(-i\frac{p^2\Delta t}{2m}\right)\psi(p) \end{cases}$$

So, we can apply them by switching between the two representations by means of the Fourier transform $\mathcal{F}$, ending up with

$$|\psi(x, t + \Delta t)\rangle = e^{-\frac{iV\Delta t}{2}}\mathcal{F}^{-1}e^{-iT\Delta t}\mathcal{F}e^{-\frac{iV\Delta t}{2}}\,|\psi(x,t)\rangle\,.$$

The main advantage of this technique is that – by using the correct representations – we are able to use diagonal operators, reducing the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. Moreover, Fourier transforms (which usually imply matrix-vector multiplication, $\mathcal{O}(N^2)$) can be efficiently computed using fast Fourier transform algorithms, which scale as $\mathcal{O}(N \log N)$.

2 Code development

We are interested in the time evolution of the ground state wave function $|\psi_0\rangle$. The main program is structured as follows:

- define the Hamiltonian H. This is done using the DiscretizedLaplacian and HarmonicPotential subroutines;
- diagonalize H using the ComputeEigenvalues subroutine, which is a wrapper for the LAPACK zheev subroutine. Notice that we use the info flag to ensure that the computation has been successful;
- take the first eigenvector as $|\psi_0(t = 0)\rangle$;
- compute the time evolution using the subroutine psiTimeEvol. In particular, we compute the evolution of the wave function, of the corresponding probability and of the potential V;
- compute the evolution of the mean $\langle x\rangle$;
- save the results in different output files.

The DiscretizedLaplacian, HarmonicPotential and ComputeEigenvalues subroutines can be found in the HarmonicOscillator1D module, psiTimeEvol in the TimeEvolution module and the subroutines to write the results in the Utilities module.

The split-operator method is implemented inside the psiTimeEvol subroutine.

```fortran
subroutine psiTimeEvol(psi_t,prob_t,x_grid,p_grid,t_len,dt,omega,m,hbar,T,V_t,dx)
    complex*16, dimension(:,:) :: psi_t
    real*8, dimension(:,:) :: prob_t, V_t
    complex*16, dimension(:), allocatable :: tmp
    complex*16, dimension(:), allocatable :: evol_op_V, evol_op_T
    real*8, dimension(:) :: x_grid, p_grid
    real*8 :: t_i, dt, omega, m, hbar, T, q0, dx, dp
    integer :: t_len, ii
    allocate(evol_op_V(t_len))
    allocate(evol_op_T(t_len))
    allocate(tmp(size(psi_t, 1)))
    dp = (2*4.d0*datan(1.d0)) / (dx*size(psi_t, 1))
    tmp = psi_t(:, 1)
    call NormalizePsi(tmp, dx)
    evol_op_T = exp( - (dt*dcmplx(0.d0,(p_grid)**2)) / (2*m*hbar) )
    do ii = 1, t_len-1
        t_i = ii * dt
        q0 = t_i / T
        evol_op_V = exp(-(dt*m*dcmplx(0.d0,(omega**2)*((x_grid-q0)**2)))/(4*hbar))
        ! apply the position space operator
        tmp = tmp * evol_op_V
        ! flip to momentum space using the FT
        tmp = FourierTransform(tmp)
        call NormalizePsi(tmp, dp)
        ! apply the momentum space operator
        tmp = tmp * evol_op_T
        ! flip to position space using the IFT
        tmp = InverseFourierTransform(tmp)
        call NormalizePsi(tmp, dx)
        ! apply again the position space operator
        tmp = tmp * evol_op_V
        call NormalizePsi(tmp, dx)
        ! store the results
        psi_t(:, ii+1) = tmp
        prob_t(:, ii+1) = abs(tmp)**2
        V_t(:, ii+1) = 0.5*m*(omega**2)*((x_grid-q0)**2)
    end do
    deallocate(evol_op_T, evol_op_V, tmp)
    return
end subroutine psiTimeEvol
```

Notice that the normalization of the wave function is constantly checked using the `NormalizePsi` subroutine, which can be found in the `Utilities` module.

The computation of the discrete Fourier transform and inverse Fourier transform is carried out using the FFTW library. In particular, since we want to compute one-dimensional Fourier transforms, we use the subroutine `dfftw_plan_dft_1d`.

```
function FourierTransform(array) result(ft)
    complex*16, dimension(:) :: array
    complex*16, dimension(size(array)) :: ft
    integer*8 :: n
    ! create a plan
    call dfftw_plan_dft_1d(n, size(array), array, ft, FFTW_FORWARD, FFTW_ESTIMATE)
    ! compute the actual transform
    call dfftw_execute_dft(n, array, ft)
    ! deallocate
    call dfftw_destroy_plan(n)
    ft = ft / sqrt(real(size(array)))
    return
end function FourierTransform

function InverseFourierTransform(array) result(ift)
    complex*16, dimension(:) :: array
    complex*16, dimension(size(array)) :: ift
    integer*8 :: n
    ! create a plan
    call dfftw_plan_dft_1d(n, size(array), array, ift, FFTW_BACKWARD,
    FFTW_ESTIMATE)
    ! compute the actual transform
    call dfftw_execute_dft(n, array, ift)
    ! deallocate
    call dfftw_destroy_plan(n)
    ift = ift / sqrt(real(size(array)))
    return
end function InverseFourierTransform
```

Particular attention must be paid when using the `dfftw_plan_dft_1d` subroutine. In fact, since the Discrete Fourier Transform is periodic in its transformed variable the positive frequencies are stored in the first half of the output and the negative frequencies are stored in backwards order in the second half of the output. This means that we have to perform some sort of "folding" of the domain, which is done computing the momentum grid in the following way.

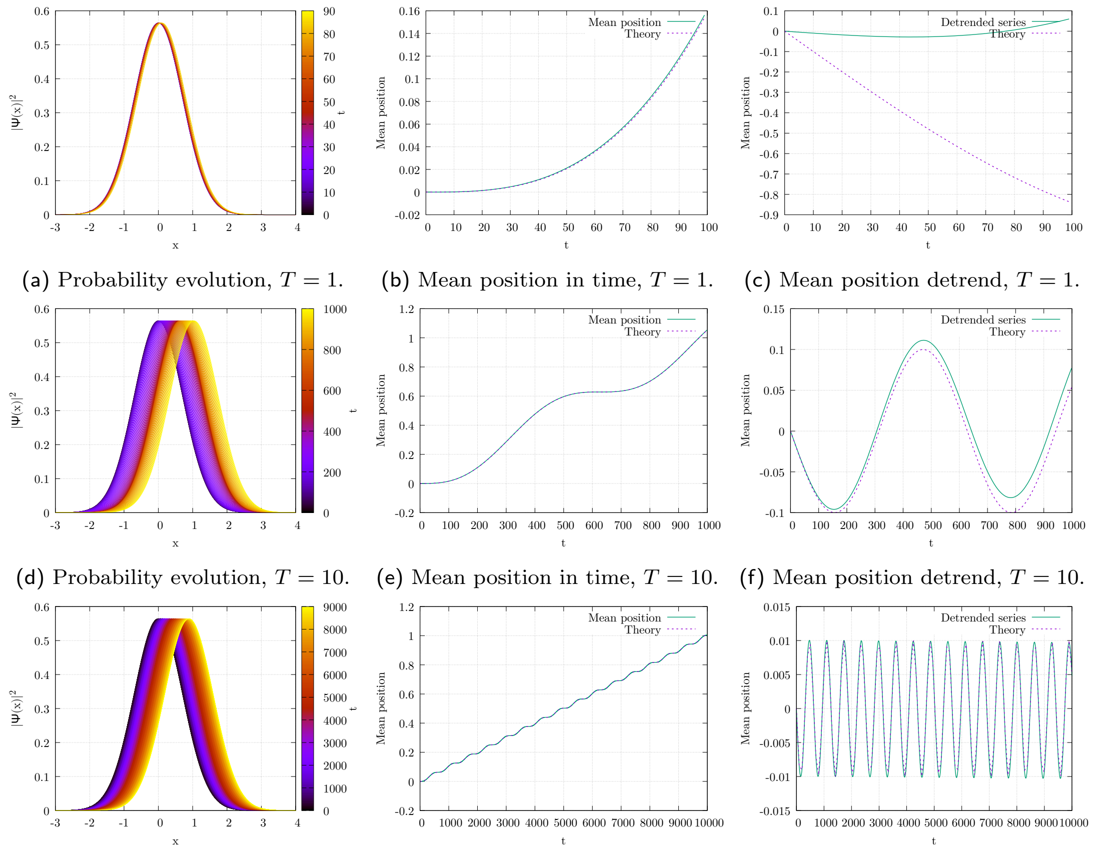
```
    do jj = 1, N/2
        p_grid(jj) = (jj-1) * (2*pi)/(dx*N)
    end do
    do jj = N/2+1, N
        p_grid(jj) = (jj-1-N) * (2*pi)/(dx*N)
    end do
```

Also, notice that FFTW computes an unnormalized transform, in that there is no coefficient in front of the summation in the discrete Fourier transform. In other words, applying the forward and then the backward transform will multiply the input by N (the size of the vector). For this reason, inside our functions we divide the results by $\sqrt{N}$.

When the program is executed, the user will be asked to enter the value of T. Then, the execution proceeds automatically. The output files are named according the value of T using the following notation: `data_time_evol_T.txt` (with `data` = `prob`, `psi_real`, `psi_imag`, `pr_mean`).

3 Results

The time evolution of the probability density for different values of T is shown in Figure 1. The values of the other parameters are $\hbar = m = \omega = 1$, $L = 500$ (with $N = 2L + 1$).



(a) Probability evolution, $T = 1$. (b) Mean position in time, $T = 1$. (c) Mean position detrend, $T = 1$.

(d) Probability evolution, $T = 10$. (e) Mean position in time, $T = 10$. (f) Mean position detrend, $T = 10$.

(g) Probability evolution, $T = 100$. (h) Mean position in time, $T = 100$. (i) Mean position detrend, $T = 100$.

Figure 1: Time evolution of the probability density for different values of T. The values of the other parameters are $\hbar = m = \omega = 1$, $L = 500$.

As we can see, during the evolution the state remains normalized. This makes sense, because the evolution operator is unitary. Moreover, notice that the motion is not stationary (as the motion of the potential), in fact at the beginning of the evolution the lines are more dense. This can be clearly seen by looking at Figure 1d, where we see that the lines are more dense at the beginning (in purple) and at $t \sim 500$ (in red). This is better understood if we look at Figure 1e, which presents a sinusoidal behaviour in the position of the mean, where there is a local maximum for $t \sim 500$.
For $T = 1$ this behaviour is not visible (or better, we can only see the thickening at the beginning) because there are not enough time steps. Conversely, for $T = 100$ the oscillations are very visible (Figure 1h).

If we want to understand the origin of this oscillatory behaviour, we can look at the gif in the exercise folder. We see that, at the beginning, the potential moves but the probability density do not. In fact, it begins to move after a delay and furthermore it does not move uniformly, but

it accelerates.
The physical explanation is the following. As we can see, the ground state is not at zero energy at $t = 0$, in fact it has energy $E_0 = \hbar\omega/2$. As time goes by, the potential increases on the left and decreases on the right. At a certain point the wave has a very high potential and it starts to move, even surpassing the potential. Then, the potential on the right is higher and the wave starts to decelerate until the potential on the left gets higher in turn and the process starts again.

The "Theory" line which is shown in the plots is given by the following expression for the average position:

$$\langle x \rangle = \frac{t}{T} - \frac{1}{\omega T} \sin(\omega t).$$

Notice that in the detrended series, which are obtained by fitting the data with a straight line $f(t) = at$ and then subtracting it, the agreement between our results and the theory increases with T: this happens because the for small T there are few points to fit, leading to worse results.

4 Self evaluation

This exercise was very instructive since it made me learn how to use a new library, FFTW.

One possible improvement could be to better automate the storing of the results, for example organizing them in different folders according to the value of T using a Python script.

P.S. Some of the output files are not present in the homework directory because they are very large ($\sim$ 200 MB).