Numerical Quantum Optics PHYS686 Project 2

Mark Rodenberger, Nima Dinyari, and Xiaolu Cheng

May,2007

Problem 1

In this problem we will simulate the unraveling of the stochastic master equation (SME). Here, we considered a two-level atom interacting with a monochromatic field. The output is monitored by homodyne detection of the X_1 quadrature. The (SME) is:

$$d\rho = -\frac{i}{\hbar}[H,\rho]dt + \Gamma\mathcal{D}[\sigma]\rho dt + \sqrt{\Gamma}\mathcal{H}[\sigma]\rho dW \qquad (1.1)$$

where the Lindblad superoperator,$\mathcal{D}[\sigma]\rho$, is:

$$\mathcal{D}[\alpha]\rho = \alpha\rho\alpha^\dagger - \frac{1}{2}\left(\alpha^\dagger\alpha\rho + \rho\alpha^\dagger\alpha\right), \qquad (1.2)$$

and the measurement superoperator is:

$$\begin{aligned}\mathcal{H}[\alpha]\rho &:= \alpha\rho + \rho\alpha^\dagger - Tr[\alpha\rho + \rho\alpha^\dagger]\rho \\ &= \alpha\rho + \rho\alpha^\dagger - \langle\alpha + \alpha^\dagger\rangle\rho. \qquad (1.3)\end{aligned}$$

It can be shown that (1.1) is equivalent to the stochastic Schrödinger equation (SSE):

$$d|\psi\rangle = -\frac{i}{\hbar}H|\psi\rangle dt - \frac{\Gamma}{2}\left[\sigma^\dagger\sigma - \langle\sigma+\sigma^\dagger\rangle\sigma + \frac{1}{4}\langle\sigma+\sigma^\dagger\rangle^2\right]|\psi\rangle dt + \sqrt{\Gamma}\left[\sigma - \frac{1}{2}\langle\sigma+\sigma^\dagger\rangle\right]|\psi\rangle dW. \qquad (1.4)$$

Showing the equivalence of the SSE and the SME is the only "exercise" for problem 1. The rest of the problem is to generate a program that computes quantum trajectories for this unraveling. Also, we should make some representative plots of the trajectories and verify numerically that an average over many trajectories(ie. and ensemble average) will converge to the unconditioned result.

We begin by noting that:

$$\rho = |\psi\rangle\langle\psi|. \qquad (1.5)$$

By expanding $|\psi\rangle$ to second order:

$$|\psi\rangle \to |\psi\rangle + d|\psi\rangle \qquad (6)$$

$d\rho$ becomes:

$$d\rho = d(|\psi\rangle)\langle\psi| + |\psi\rangle d(\langle\psi|) + d(|\psi\rangle)d(\langle\psi|). \tag{1.7}$$

It can be shown that in (1.7) $d\langle\psi|$ is:

$$d\langle\psi| = \frac{i}{\hbar}\langle\psi|Hdt - \frac{\Gamma}{2}\langle\psi|\left[\sigma^\dagger\sigma - \sigma^\dagger\langle\sigma+\sigma^\dagger\rangle + \frac{1}{4}\langle\sigma+\sigma^\dagger\rangle^2\right]dt + \sqrt{\Gamma}\langle\psi|\left[\sigma^\dagger - \frac{1}{2}\langle\sigma+\sigma^\dagger\rangle\right]dW. \tag{1.8}$$

With these definitions we can compute $d\rho$:

$$\begin{aligned} d\rho = &-\frac{i}{\hbar}H\rho dt - \frac{\Gamma}{2}\left[\sigma^\dagger\sigma - \langle\sigma+\sigma^\dagger\rangle\sigma + \frac{1}{4}\langle\sigma+\sigma^\dagger\rangle^2\right]\rho dt + \sqrt{\Gamma}\left[\sigma - \frac{1}{2}\langle\sigma+\sigma^\dagger\rangle\right]\rho dW \\ &+\frac{i}{\hbar}\rho H dt - \frac{\Gamma}{2}\rho\left[\sigma^\dagger\sigma - \sigma^\dagger\langle\sigma+\sigma^\dagger\rangle + \frac{1}{4}\langle\sigma+\sigma^\dagger\rangle^2\right]dt + \sqrt{\Gamma}\rho\left[\sigma^\dagger - \frac{1}{2}\langle\sigma+\sigma^\dagger\rangle\right]dW \\ &+\Gamma\left[\sigma - \frac{1}{2}\langle\sigma+\sigma^\dagger\rangle\right]\rho\left[\sigma^\dagger - \frac{1}{2}\langle\sigma+\sigma^\dagger\rangle\right](dW)^2. \end{aligned} \tag{1.9}$$

To get to this result one should only keep terms on the order of dt, while keeping in mind that $(dW)^2$ is of the same order of dt. With this we get:

$$d\rho = -\frac{i}{\hbar}[H,\rho]dt + \left(-\frac{\Gamma}{2}\sigma^\dagger\sigma\rho - \frac{\Gamma}{2}\rho\sigma^\dagger\sigma + \Gamma\sigma\rho\sigma^\dagger\right)dt + \sqrt{\Gamma}\left(\sigma\rho + \rho\sigma^\dagger - \rho\langle\sigma+\sigma^\dagger\rangle\right)dW. \tag{1.10}$$

which is the SME.

In terms of actual coding for this problem we had to modified the sample code in the following ways:
First, we added the module globals.f90, in which we declare some new parameters that could be defined and used among the different modules. Second, we had to change the equations of motion to that of (1.4) in the sderk_support.f90 file to solve the SSE.

In doing so we represented $|\psi\rangle$ as a 2×1-matrix of the form:

$$|\psi\rangle = \begin{pmatrix} C_e \\ C_g \end{pmatrix}$$

which is used in the equations of motion.

We compared the outputs of a different number of substeps by taking the difference of the solutions for the two. Obviously adding more substeps increase the time it takes for the integrator. Therefor, we tried to find a number of substeps that would give us an accurate enough of a solution without increasing the simulation time by too much.

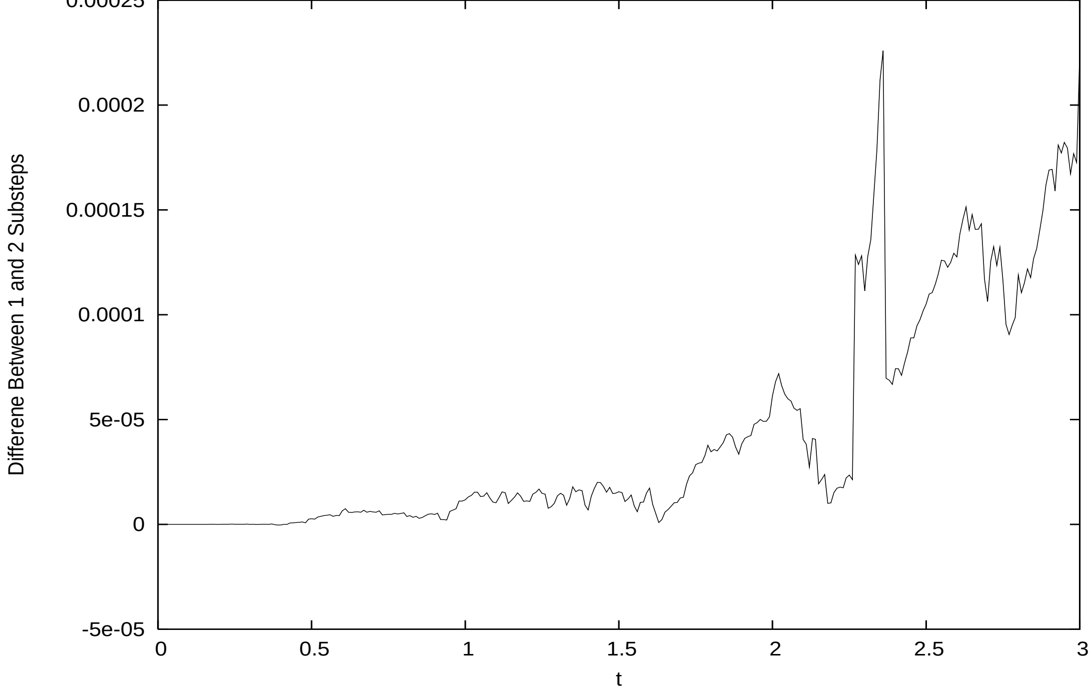

Fig. 1.1: This shows the difference between one and two substeps is taken on the same Weiner path for the SSE.

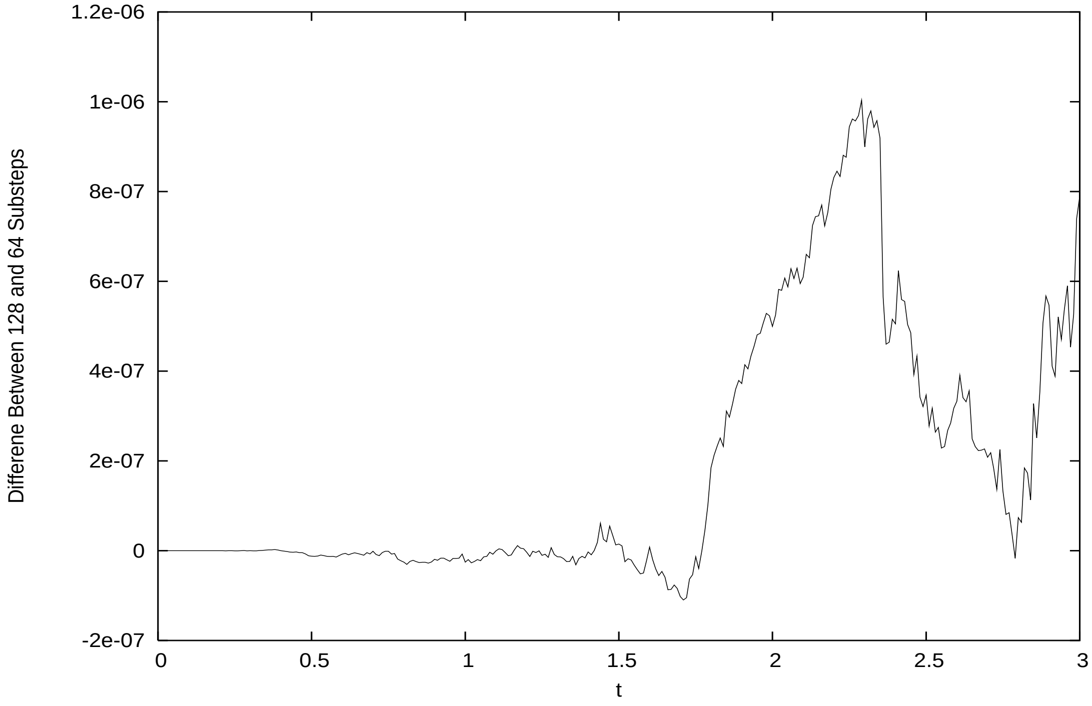

Fig. 1.2: This shows the difference between 128 and 64 substeps is taken on the same Weiner path for the SSE.

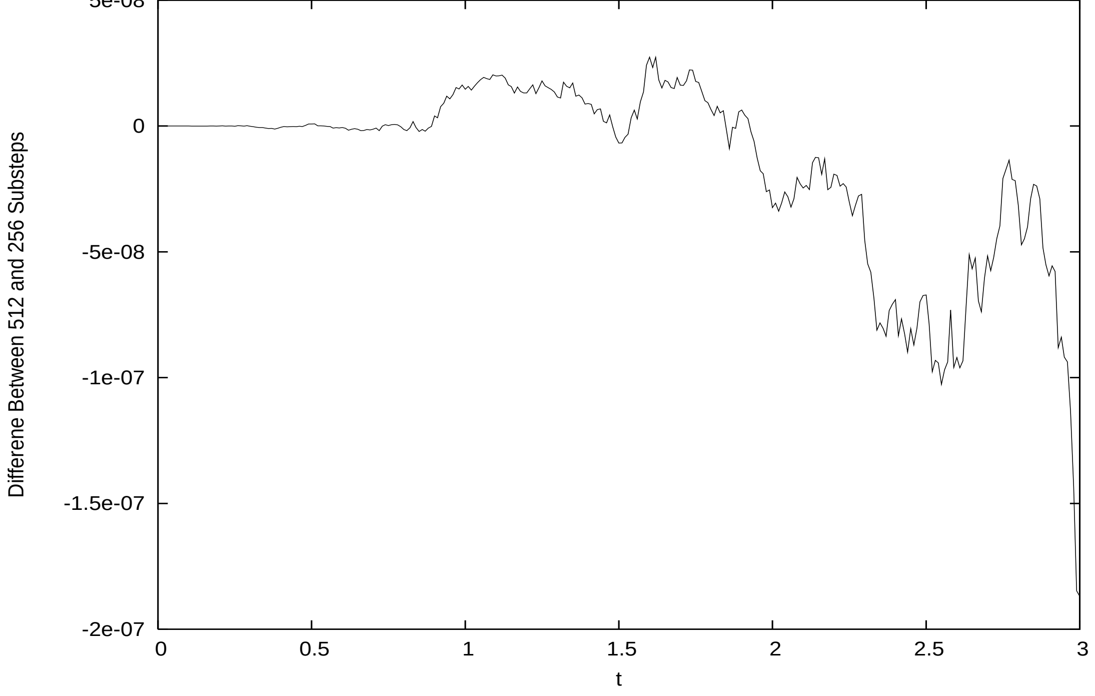

Fig. 1.3: This shows the difference between 512 and 256 substeps is taken on the same Weiner path for the SSE.

We made many more plots but these three will suffice for the following explanaition:

In comparing the different number of substeps we found that increasing the number of substeps only changed the difference between the generated solutions by 10ˆ3 in error. So we determined that either 64 or 128 substeps will do. Having too few will result in a bigger difference with a larger number of substeps, while too many will increase the run time by too much as mentioned above. Also, it should be noted that this is for only one trajectory, so the run time for more substeps with many trajectories will increase the run time significantly.

Finally, below we have a plot of the solution for ρ_{ee} for a number of trajectories initially in the ground sate, with $\Omega = 10$, $\Gamma = 1$, and $\Delta = 0$. We had to modify the code such that it could preform this operation. It was simply a loop that added up many trajectories and then averaged them after the loop completed. This loop is exactly the same as the ones in the first project. As one can see, when you add enough trajectories the solution to the stochastic equation of motion matches that of the analytical solution.

The hash marks are the analytical solution. The thin solid line is for one trajectory. The thin dashed line is for 100 trajectories; it fits pretty until the end. The thick solid line represents 10,000 trajectories and as you can see fits pretty well.

So this concludes the discussion for problem one. At the end of this assignment is the code for the problem. The code for this problem can also be found in group1's folder, in project2/problem1. The different plots for this problem can be found in project2/problem1/figuresfor1 where the plots not used in this write-up are not in the use directory.

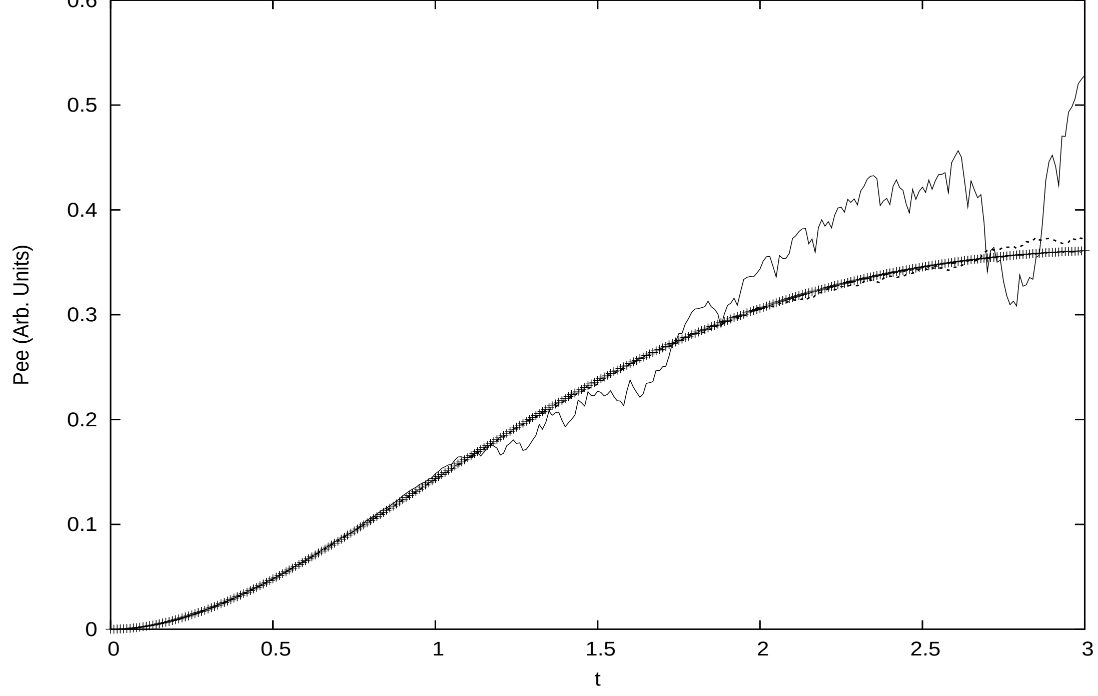

Fig. 1.4: This plot shows the evolution of ρ_{ee} as a function of time for 1 trajectory (thin solid), 100 trajectories(thin dashed), 10,000 trajectories(thick solid), and the analytical(hash marks).

Code for globals and sderk_support that were modified for this problem:

0.1 sde

```
!-----------------------------------------------------------------------
! !  Globals module.  Put all variables to be accessible everywhere
!    here. We basically added this from the first project since we
wanted !    to use variables that would be used in both the
equations of motion !    and to be used in sderk_support.f90 and
sdesample.f90. The difference !    here from project 1 is that we
have added the hamiltonian and the !    lowering operator as complex
matricies. !
!-----------------------------------------------------------------------

module globals

  ! this sets the floating-point precision to double; basically
  ! it says "give me the precision needed to get 14 decimal places,
  ! and call it wp".  From now on, we declare variables to be of
  ! type wp.  You can change to "p=4" for single precision, but
  ! for now to be safe we will stick to doubles.

  integer, parameter  :: sderk_prec = selected_real_kind(p=14)
  integer, parameter  :: wp = sderk_prec

  real(wp) :: Omega       ! Rabi Frequency
  real(wp) :: Omega_gamma ! Rabi frequencey in presence of damping
  real(wp) :: delta       ! Detuning
  real(wp) :: gamma       !decay rate

  complex(wp), dimension(2,2) :: sigma, Ham !lowering operator
                                  !and Hamiltonian

  ! 'parameter' here means that the value of the variable can't be changed
  complex(wp), parameter :: i = (0.0_wp, 1.0_wp)  ! sqrt (-1)

end module globals



!-----------------------------------------------------------------------
!  This is the support module that is used with sderk90.f90 and !
when ./sdesample ! !  This is where the equations of motion are for
the problem that is !  to be integrated. It is currently set up to
integrate the SSE of !  problem1. There are two parts to the
equation. One is the !  deterministic part, the first one, and the
second one is the !  stochastic part. ! !
!-----------------------------------------------------------------------

module sderk_support

  use globals

  ! set precision of calculation to double precision
  !integer, parameter  :: sderk_prec = selected_real_kind(p=14)
  !integer, parameter  :: wp = sderk_prec

  ! use implicit order 1.5 method for integration; just leave the tolerance
  integer, parameter  :: sderk_mf = 23
  real(wp), parameter :: sderk_tol = epsilon(1.0_wp) * 100

  ! use good random number generator, just leave this
  integer, parameter  :: sderk_rand_mf = 301

  ! declare storage; you may need to change to complex or the dimension
  complex(wp), dimension(2), save, target :: sderk_y
  complex(wp), dimension(2), save ::                                    &
           sderk_a, sderk_b, sderk_c, sderk_ybar,                     &
           sderk_aybarp, sderk_aybarm, sderk_bybarp, sderk_bybarm,     &
           sderk_aphip, sderk_aphim, sderk_bphip, sderk_bphim



contains


! Subroutines that implement equations of motion, to be called !
by sderk; note that you need to supply *two* routines, !   one to
return the dt part and one to return the dW part.

subroutine sderk_func_a(t, y, ydot)
  implicit none
  real(wp) :: t
  complex(wp), dimension(2) :: y, ydot

  ! this is the Ito form
  ! return deterministic derivative
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! !   We obviously had to change the equations of motion to the
stocahstic !   schrodinger equation which is what we want to solve.
Like before, !   Below is the deterministic part. ! !   The full
equation of motion is given in the XXXXXXXX module ! !
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


  ydot = -i*matmul(Ham,y) -gamma*matmul(matmul(transpose(sigma),sigma)&
        &-dot_product(y,matmul(sigma+transpose(sigma),y))*sigma,y)*0.5_wp&
        &-gamma*dot_product(y,matmul(sigma+transpose(sigma),y))**2.0_wp&
        &*0.125_wp*y


  return
end subroutine sderk_func_a

subroutine sderk_func_b(t, y, ydot)
  implicit none
  real(wp) :: t
  complex(wp), dimension(2) :: y, ydot

  ! return stochastic derivative
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! !   The stochastic part of the equation of motion had to be change
too !
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


  ydot = sqrt(gamma)*(matmul(sigma,y)&
        &-dot_product(y,matmul(sigma+transpose(sigma),y))*y*0.5_wp)

  return

end subroutine sderk_func_b


end module sderk_support
```

Problem 2

In this problem, we will evolve the solution of the time-dependent Schrödinger equation for a particle in a harmonic potential well. The equation of motion is:

$$i\hbar\partial_t\psi(x,t) = H\psi(x,t) = \left(\frac{p^2}{2} + \frac{1}{2}\omega^2 x^2\right)\psi(x,t)$$

where the coordinate x and the momentum p are mass-weighted. As is well known, the evolution of the wave function over a time Δt will be:

$$\psi(x, t+\Delta t) = e^{-iH\Delta t/\hbar}\psi(x,t)$$

In this problem we will use the split-operator exponential method which is to split the Hamiltonian into two parts, each of which is diagonal in their respective representations.

$$H = T(p) + V(x)$$

Noting the Baker-Campbell-Hausdorff expansion we have:

$$e^A e^B = exp\left(A + B\frac{1}{2}[A,B] + \frac{1}{12}[A,[A,B]] + \frac{1}{12}[[A,B],B] + \dots\right)$$

Then the time-evolution operator will be split as

$$e^{-iT(p)\Delta t/\hbar}e^{-iV(x)\Delta t/\hbar} = e^A$$

where A is:

$$A = -iH\Delta t/\hbar + \frac{1}{2}(-i\Delta t/\hbar)^2[T,V] + \frac{1}{12}(-i\Delta t/\hbar)^3[T,[T,V]] + \frac{1}{12}(-i\Delta t/\hbar)^3[[T,V],V]$$

We can then expand the time-evolution operator to an approximate one with error of Δt^2:

$$e^{-iT(p)\Delta t/\hbar}e^{-iV(x)\Delta t/\hbar} = e^{-iH\Delta t/\hbar} + \emptyset(\Delta t^2)$$

An even more accurate method is to split it the time-evolution operator symmetrically with respect to the potential term:

$$e^{-iV(x)\Delta t/2\hbar}e^{-iT(p)\Delta t/2\hbar}e^{-iV(x)\Delta t/2\hbar} = e^{-iV(x)\Delta t/2\hbar}e^{B}$$

where B is:

$$B = -i(T+V/2)\Delta t/\hbar + \frac{1}{2}(-i\Delta t/\hbar)^2[T,V/2] + \frac{1}{12}(-i\Delta t/\hbar)^3[T,[T,V/2]] + \frac{1}{12}(-i\Delta t/\hbar)^3[[T,V/2],V/2]$$

We can also write this as:

$$e^{-iV(x)\Delta t/2\hbar}e^{-iT(p)\Delta t/2\hbar}e^{-iV(x)\Delta t/2\hbar} = e^{C}$$

where C is:

$$C = -iH\Delta t/\hbar + \frac{1}{2}(-i\Delta t/\hbar)^2[T, V/2] + \frac{1}{12}(-i\Delta t/\hbar)^3[T,[T,V/2]] + \frac{1}{12}(-i\Delta t/\hbar)^3[[T,V/2],V/2]$$

$$+\frac{1}{2}(-i\Delta t/\hbar)^2[V/2, T+V/2] + \frac{1}{4}(-i\Delta t/\hbar)^3[V/2,[T,V/2]] + \frac{1}{24}(-i\Delta t/\hbar)^4 \ldots$$

The terms to the order Δt^2 cancel each other and therefor the error in this splitting goes as Δt^3:

$$e^{-iV(x)\Delta t/2\hbar}e^{-iT(p)\Delta t/2\hbar}e^{-iV(x)\Delta t/2\hbar} = e^{-iH\Delta t/\hbar} + \emptyset(\Delta t^3)$$

To evolve the wave function the program will run something like this. After initializing all of the parameters and opening all of the files where the output will be stored it then evolves the wavefunction. As an example, we will explain the second order method and the other orders will follow suit.

First we will evolve the wavefunction in the position representation by applying $e^{-iV(x)\Delta t/2\hbar}$, with half the time step, in the x-representation where $V(x)$ is diagonal. Second, the program will Fourier Transform the wavefunction into the momentum representation the wavefunction can be evolved by applying $e^{-iT(p)\Delta t/2\hbar}$ where $T(p)$ is diagonal. Finally, the program will Fourier Transform the wavefunction back to the position representation and repeat the first step. This is the basic idea of the evolution. Thus time evolution over one time step Δt is done.Then we can repeat this procedure for many time steps.

Here is the answer to one of the exercises where you ask to show that if the two splittings, (12) and (13) in the handout, have approximately the same error after many steps:

The local error for each step goes like Δt^3 in the symmetric splitting. When this factor is multiplied by a number of steps N time steps, which is of order $\frac{1}{\Delta t}$, we get a global error which goes as Δt^2. After many steps, the global error according to the two splitting methods is almost the same. This can be seen from using the symmetric splitting, after two steps:

$$e^{-iV(x,t+\Delta t)\Delta t/2\hbar}e^{-iT(p,t+\Delta t)\Delta t/\hbar}e^{-iV(x,t+\Delta t)\Delta t/2\hbar}e^{-iV(x,t)\Delta t/2\hbar}e^{-iT(p,t)\Delta t/\hbar}e^{-iV(x,t)\Delta t/2\hbar}$$

In which $V(x, t+\Delta t)$ is approximately $V(x,t)$, then the above becomes:

$$e^{-iV(x,t+\Delta t)\Delta t/2\hbar}e^{-iT(p,t+\Delta t)\Delta t/\hbar}e^{-iV(x,t)\Delta t/\hbar}e^{-iT(p,t)\Delta t/\hbar}e^{-iV(x,t)\Delta t/2\hbar}$$

The middle part is just like the asymmetric splitting method. If we keep combining terms like this, we find that the final evolution of wave function using these two splitting is approximately the same.

A nice method called Richardson extrapolation is used to reduce the error to even higher orders. Keep in mind that, for each time step Δt of the evolution, we will introduce an error of Δt^3, then after N steps:

$$\psi(x,N\Delta t)=\left(e^{-iH\Delta t/\hbar}+\emptyset(\delta t^3)\right)^N\psi(x,0)$$

where the coefficient is:

$$\left(e^{-iH\Delta t/\hbar}+\emptyset(\delta t^3)\right)^N=e^{-iH\Delta t/\hbar}+Ne^{-iH\Delta t(N-1)}\emptyset(\Delta t^3)$$
$$+N(N-1)/2e^{-iH\Delta t(N-2)}\emptyset(\Delta t^6)+N(N-1)(N-2)/6e^{-iH\Delta t(N-3)}\emptyset(\Delta t^9)+\ldots.$$

We know the number of time steps is total time divided by time step size, which is therefor proportional to Δt^{-1}. Then:

$$N^2\emptyset(\Delta t^6)\sim\emptyset(\Delta t^4)$$
$$N^3\emptyset(\Delta t^9)\sim\emptyset(\Delta t^6)$$

and so on. Hence we find that only even terms will appear; answering another exercise. We can then take the Richardson's extrapolation as an ansatz that the global error of the numerical solution takes the form:

$$\tilde{\psi}_{\Delta t}(x,t)-\psi(x,t)=e_2(x,t)\Delta t^2+e_4(x,t)\Delta t^4+e_6(x,t)\Delta t^6+\ldots$$

Then we can compute the solutions with multiple step size, and add different linear combinations of them to cancel the low order terms of the error.

The fourth-order method follows from using step size of Δt and $\Delta t/2$:

$$\frac{4}{3}\tilde{\psi}_{\Delta t/2}(x,t)-\frac{1}{3}\tilde{\psi}_{\Delta t}(x,t)=\frac{4}{3}\psi(x,t)+\frac{4}{3}e_2(x,t)(\frac{\Delta t}{2})^2+\frac{4}{3}e_4(x,t)(\frac{\Delta t}{2})^4+\ldots$$
$$-\frac{1}{3}\psi(x,t)-\frac{1}{3}e_2(x,t)\Delta t^2-\frac{1}{3}e_4(x,t)\Delta t^4-\ldots$$
$$=\psi(x,t)-\frac{1}{4}e_4(x,t)\Delta t^4+\ldots.$$

The sixth-order method follows from using step sizes of Δt, $\Delta t/2$ and $\Delta t/3$:

$$\frac{1}{24}\tilde{\psi}_{\Delta t}(x,t)-\frac{16}{15}\tilde{\psi}_{\Delta t/2}(x,t)+\frac{81}{40}\tilde{\psi}_{\Delta t/3}(x,t)=\frac{1}{24}\psi(x,t)+\frac{1}{24}e_2(x,t)\Delta t^2+\frac{1}{24}e_4(x,t)\Delta t^4+\frac{1}{24}e_6(x,t)\Delta t^6\ldots$$
$$-\frac{16}{15}\psi(x,t)-\frac{16}{15}e_2(x,t)(\frac{\Delta t}{2})^2-\frac{16}{15}e_4(x,t)(\frac{\Delta t}{2})^4-\frac{16}{15}e_6(x,t)(\frac{\Delta t}{2})^6-\ldots$$
$$+\frac{81}{40}\psi(x,t)+\frac{81}{40}e_2(x,t)(\frac{\Delta t}{3})^2+\frac{81}{40}e_4(x,t)(\frac{\Delta t}{2})^4+\frac{81}{40}e_6(x,t)(\frac{\Delta t}{2})^6+\ldots$$
$$=\psi(x,t)+(\frac{1}{24}-\frac{4}{15}+\frac{9}{40})e_2(x,t)\Delta t^2+(\frac{1}{24}-\frac{1}{15}+\frac{1}{40})e_4(x,t)\Delta t^4+(\frac{1}{24}-\frac{1}{60}+\frac{1}{360})e_6(\Delta t)^6+\ldots$$
$$=\psi(x,t)+\frac{1}{36}e_6(\Delta t)^6+\ldots$$

In terms of actually using the code for this problem we ran the program using the second-order, fourth-order, and sixth-order separately and then compared their output for different time steps to the analytical.

To do this we add a loop to 'sch1d.f90', the only editing to the program, which ran the program, took the difference between the numerical and the exact solution after two periods, and then increased the step size and ran the code again. We then compare the error of this difference after 2 periods with respect to the exact solution and plot the error as a function of step size dt.

The reason that we can do this is because ω, the frequency of the potential, is 2π, and hence the period is 1 second. If we take the difference of the numerical solution at a final time of 2 seconds with the initial conditions it is equivalent to comparing it to the exact solution after two seconds; this is true because the exact solution is periodic and therefor equal to the initial conditions after an integer multiple of periods.

When we calculate the difference of the distribution between the final time, two periods here, and at the initial time we define this as the global error. To make sure that when we keep changing dt, the final time t_final is still two periods, we increment dt by $1/5040 \times t_final$, where:

$$5040 = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7$$

.
This is a common factor of the numbers 1 through 10.

Then we added a standard output to the file such that we can have the difference of the numerical solution to the exact solution with the corresponding step size. So if we run our program like: './sch1d sample.param > out.txt', we will save all of the differences with all different time step sizes which we can then plot to see the global error as a function of step size.

Figure 2.1, 2.2, and 2.3 shows the behavior for the 2nd, 4th, and 6th order methods. There is a basic trend for all of them.

The error decreases with decreasing dt, even though all of them are as small as 10^{-11}, but after some value of dt the error begins to increase. This implies that there is some ideal dt and making it too big *or* too small will introduce error that is due to just the step size.

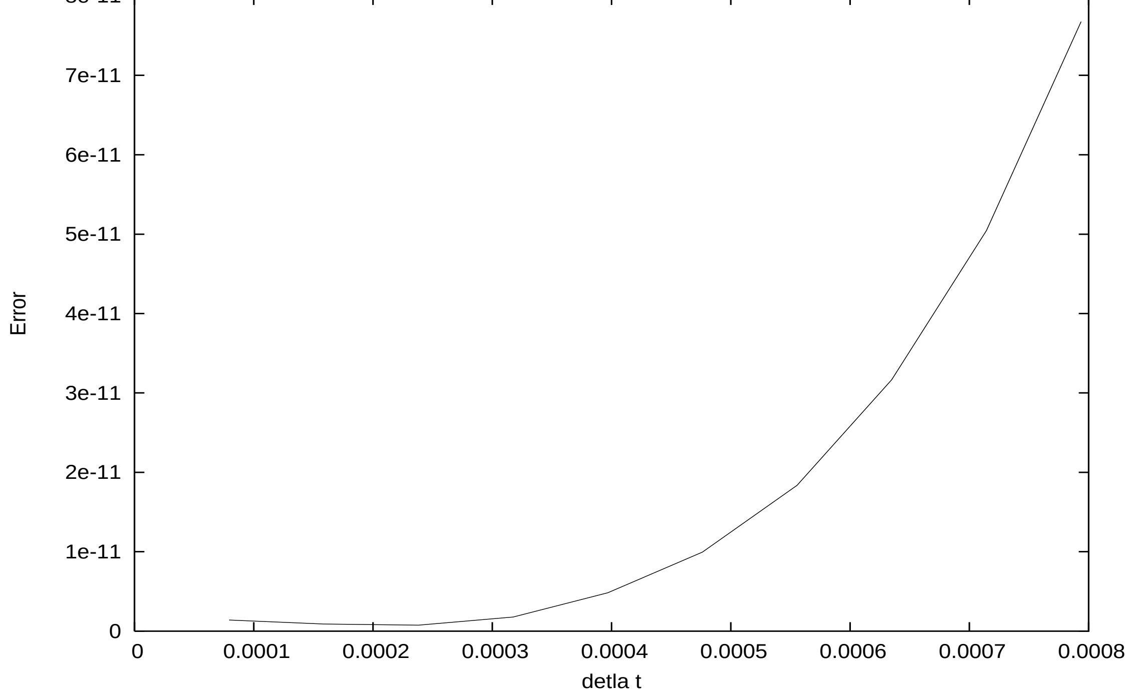

Figure 2.1: Difference between initial and final distributions using second-order method

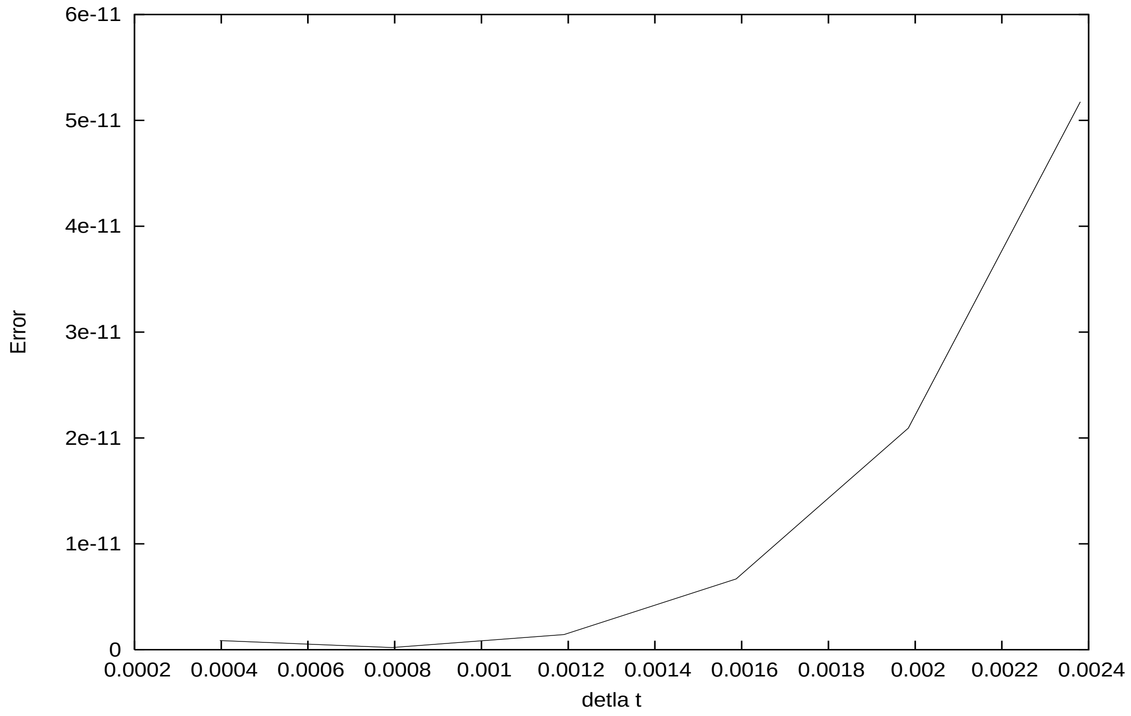

Figure 2.2: Difference between initial and final distributions using fourth-order method

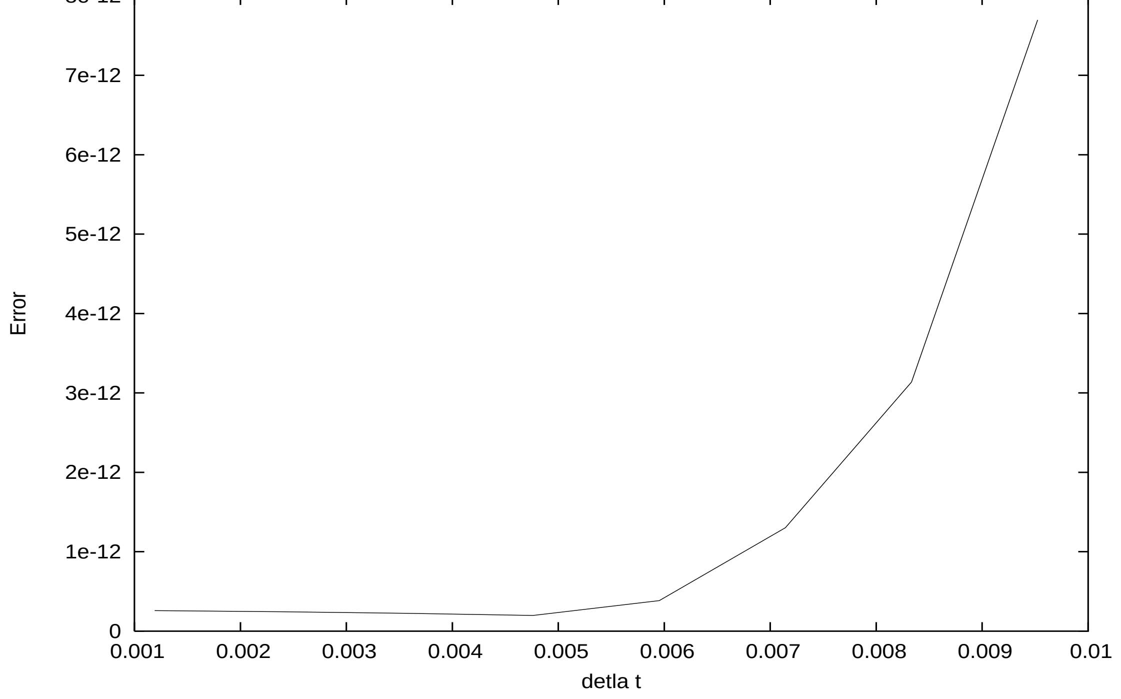

Figure 2.3: Difference between initial and final distributions using sixth-order method

We also changed the time step size to see the behavior of $\langle x\rangle$, $\langle p\rangle$, V_x, and V_p at these different step sizes. For step sizes from $dt = 0.0000005$ to $dt = 0.01$, the behavior of each looks fine; they are oscillating normally with time. Figure 2.4 below shows their behavior for $dt = 0.0005$.

However, if we increase the step size to $dt = 0.3$, for example, the plot won't be nearly as smooth as before. If we then increase dt to 0.5, $\langle x\rangle$, $\langle p\rangle$, V_x, and V_p won't even oscillate and the functions look ridiculous.

These phenomena are reasonable because the period is 1 second, and the numerical calculations only make sense when the step size is much smaller than the period, so we can have enough sample points within one period to resolve the function.

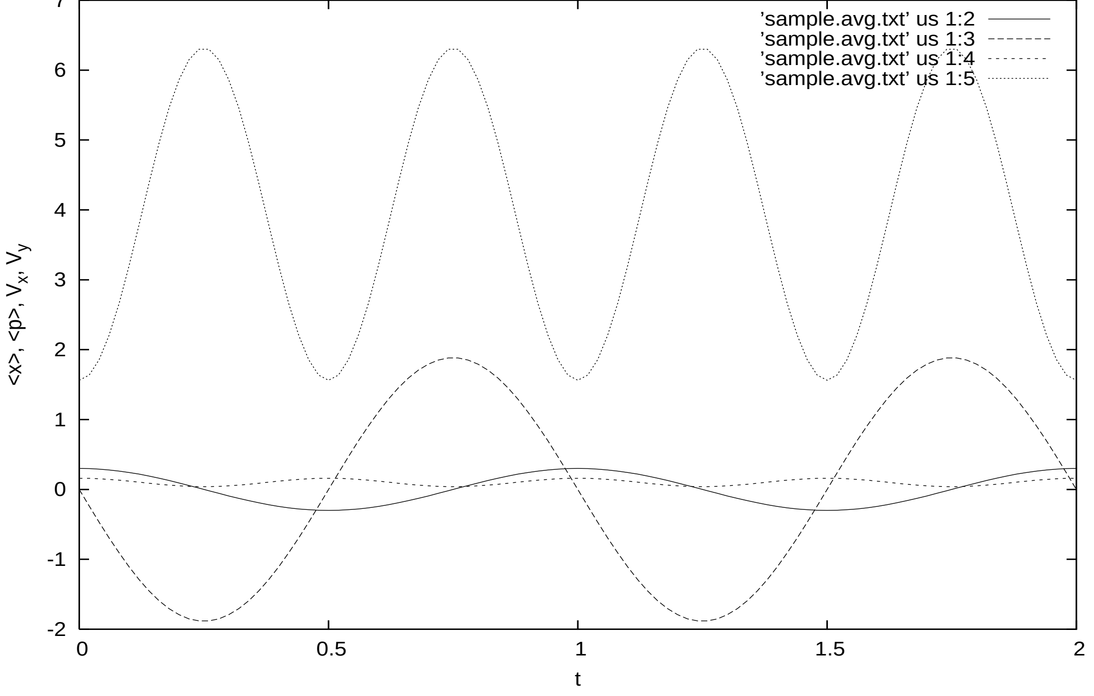

Figure 2.4: The behavior for a time step of $\Delta t = 0.0005$. The solid line represents the behavior of $\langle x \rangle$, the dashed line represents the behavior of $\langle p \rangle$, the dot-space line represents the behavior of V_x, and the small dot line represents the behavior of V_p.

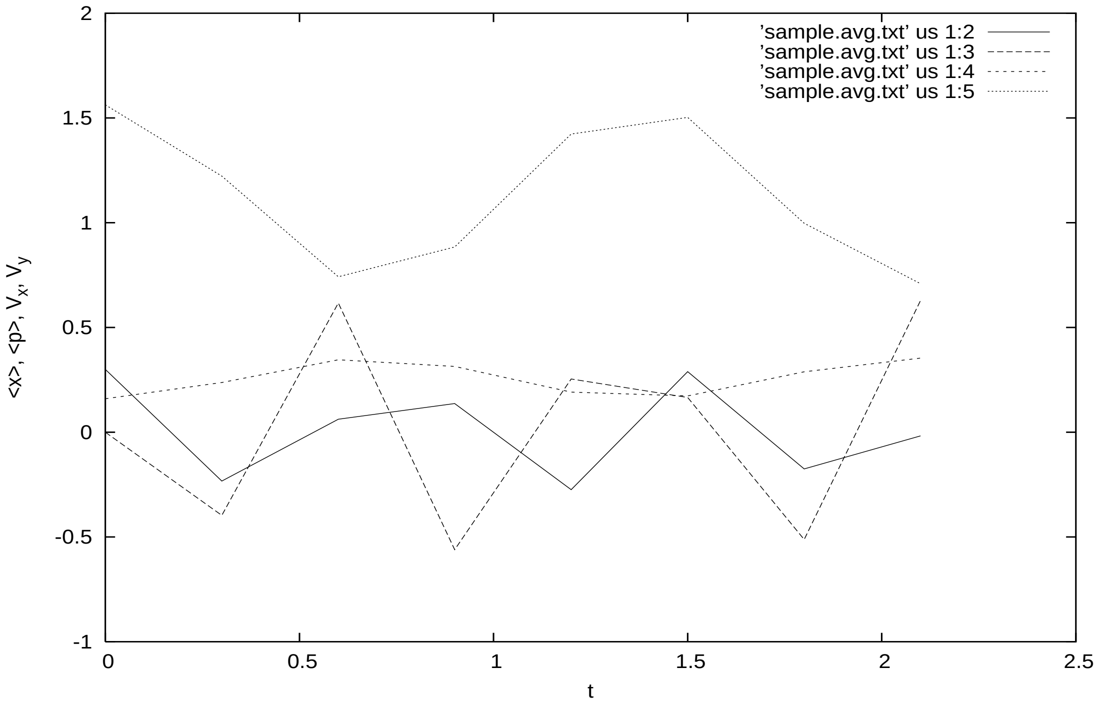

Figure 2.5: The behavior for a time step of $\Delta t = 0.3$. The solid line represents the behavior of $\langle x \rangle$, the dashed line represents the behavior of $\langle p \rangle$, the dot-space line represents the behavior of V_x, and the small dot line represents the behavior of V_p.

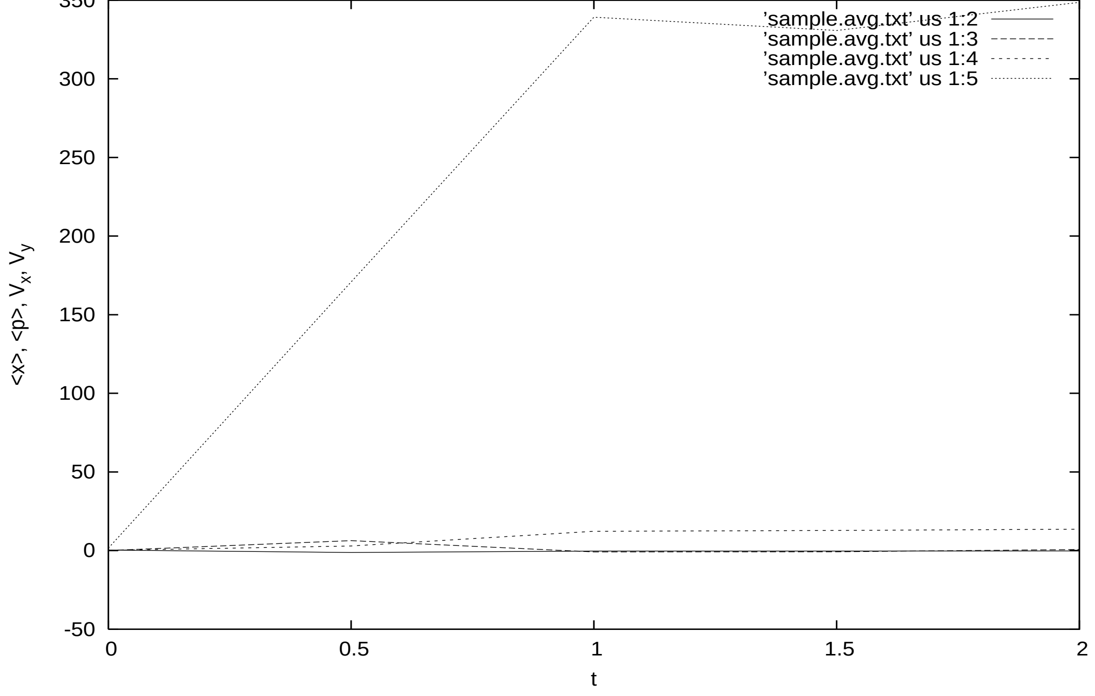

Figure 2.6: The behavior for a time step of $\Delta t = 0.5$. The solid line represents the behavior of $\langle x \rangle$, the dashed line represents the behavior of $\langle p \rangle$, the dot-space line represents the behavior of V_x, and the small dot line represents the behavior of V_p.

Code which was modified when using sch1d:

0.2 sch1d

```
                                                                           ! close output files
!=====================================================================   call cleanup()
! !  sch1d.f90 ! !  Computes evolution of Schrodinger equation in
one dimension !    by split-operator exponentials. ! !  The equation     end program sch1d
in scaled units (with unit mass) is ! !
2 !    i hbar d  psi = ( - (1/2) d  + V(x) ) psi !            t
x ! !    the potential is taken to be harmonic, ! !          2     2
!    U = (w /2) x ! !  Evolution is by explicit split-operator !
exponentials, to give a second order method extendable to !    4th,
6th, etc orders. ! !  Output is to 2 text files: !    1. .xdist.txt
!       first column is the time, the other columns are the
populations !       at each of the position-grid values !    2.
.pdist.txt !       first column is the time, the other columns are
the populations !       at each of the momentum-grid values ! !
Numerical error vs. step size is put to stdout !
!=====================================================================

program sch1d

  use globals
  use output
  use param_parser
  use splitop_evolver
  use timing

  implicit none

  integer  :: j, n
  integer  :: ti, tiout

  ! get parameters from parameter file
  call parse_params()

  ! set and save initial condition (Gaussian)
  ! initial remains, psi is evolved
  initial  = exp( -(x-x0)**2/(4*sigma_x**2) ) / sqrt(sqrt(2*pi)*sigma_x)
  psi = initial

  ! output initial condition
  call open_output_files()
  call output_stuff()
  call output_dists(first_time=.true.)

! main loop; each run with another step size do n= 1, 8

  psi = initial

  call print_norm()
  if ( time_trace ) call print_time('s')

  dt = n * tfinal / 5040_wp            ! step size; 2520 has factors 1..10
  toutevery = int(5040_wp / n + 0.4)   ! #steps adapted to reach tfinal

  ! Main evolution loop
  t = 0.0_wp
  j = 1
  do tiout = 1, Ntout     ! this is the loop over output times
    do ti = 1, toutevery  ! this is a loop over times between outputs
      t = t + dt
      j = j + 1
      call splitop_onestep()  ! this evolves psi from t to t+dt
    end do

    ! note: for a stochastic code, you may want to make many more
    !   calls to output_stuff() than to output_dists()
    call output_stuff()       ! this writes expectation values, etc
    call output_dists()       ! this writes wave functions; *always* call
                              !   *after* output_stuff()

    if ( time_trace ) then
      write(0,*) 'Wave function computed at t = ', t
      write(0,*) '  (step ', tiout, ' of ', Ntout, ')'
    end if
    call print_norm()                 ! print wave function norm to std err
    if ( time_trace ) call print_time('m')  ! print elapsed time to std err
  end do

  write(*, format1) dt, real(sum(abs(psi*conjg(psi)-initial*conjg(initial))),wp)*dx
  ! writes step size and integrated error, int( |numerical-exact| dx ), to stdout

end do
```

Problem 3

In this problem, we are trying to simulate the quantum feedback control of the motion of a quantum-mechanical particle in a harmonic potential. So we will still consider it to be the one-dimensional harmonic oscillator given in problem 2, whose Hamiltonian is:

$$H = \frac{p^2}{2m} + \frac{1}{2}m\omega^2 x^2.$$

The stochastic Schödinger equation is:

$$d|\psi\rangle = -\frac{i}{\hbar}H|\psi\rangle dt - k(x - \langle x\rangle)^2|\psi\rangle dt + \sqrt{2k}(x - \langle x\rangle)|\psi\rangle dW,$$

where the terms proportional to k or $\sqrt{k}$ model the measurement of the particle.

This is the main equation that we intend to use for the evolution. However, since there is a stochastic part to the evolution we can't just use the code in problem two. Likewise, since there is a momentum term we can't just use the code of problem one. Therefor, we will use a combination of the two.

Since in Problem 1 we integrated a similar wave function using the stochastic Schödinger equation, we will use that code, modified for this problem, to evolve the wavefunction for the first half of the time step but ignore the p^2 part. However, for problem the wavefunction was a two dimensional vector. For this problem the wave function has many elements which is trying to represent the function's value at the various positions in the potential. So we had to use the array in the equations of motion.

Then like in the Problem 2, which split the Hamiltonian into two parts and evolved the wave function separately in position and momentum representations, we will use that code to Fourier transform the wave function into momentum space, after the first half of the potential evolution, and then evolve it using an operator of the form $e^{-ip^2\Delta t/2m\hbar}$ and then do the inverse Fourier transform back to position space.

Finally, we repeat the first part of the evolution for the last half step.

This only evolves the wavefunction; and due to the measurement it heats up the system. To complete this problem we then need to implement a feedback loop to cool the oscillator by increasing ω when x is zero, and decreasing it when x is at its turning points.

Experimentally, we would measure some value, like photocurrent, which has useful information as well as some quantum noise. However, when we do the feedback, we don't want to feedback the noise. So we will apply a low pass filter to remove the noise from the measurement, and then use that signal as feedback.

In this problem, we will use the measurement record, which follows from the taking the homodyne photocurrent with the same operator and scalar replacements:

$$dy(t) = \langle x \rangle dt + \frac{dW}{\sqrt{2k}}.$$

It contains information about the mean position value and the noise. We calculate it using the evolved wave function after each time step, then use a low pass filter to filter out the noise, and manipulate the trap strength ω in an appropriate fashion. One should be able to cool the system this way.

$$dV = -\gamma V dt + \gamma dy$$

We had some problems implementing the feedback loop. We had to do many things to get it to work. We will list them as we go.

Since we were taking in information about the mean position, via the homodyne measurement, and then passing it through a low pass filter we gained a phase lag between $\langle x \rangle$ and our measurement value of the mean position. Once we picked the right time constant we saw that we had a phase lag of $\frac{\pi}{2}$. When we took the derivative of this signal we got the peaks of the signal to coincide with the peaks of the actual value of $\langle x \rangle$! When we took the second derivative we got the zeros of that function to coincide with the the maxima of $\langle x \rangle$ without as much noise as the original signal.

We then used some value of both of these functions to be the value where either the potential grow, increase ω, or have the potential drop, decrease ω; this is the bang bang method for cooling.

The evolution is very sensitive to parameters such as the step size, the measurement strength k, the time constant of the low pass filter, and the maximal and minimal values for ω when the bang bang method is implemented.

So we spent a long time playing with all of these values until we found decent cooling of the system. To see cooling we plotted $\frac{P^2}{2m}$ as a function of t. We didn't implement the cooling until after the simulation ran for 1 second. We did this so we would have one periods worth of data in the array for the first and second derivative of the signal. We will also plot the total energy as a function of time. We saw that after we started cooling the particle the total energy dropped from $27,000(Arb.Units)$ to $6,000(ArbUnits)$!

Figure 3.2 shows the kinetic energy over time. It is fluctuating due to the oscillatory behavior of the system. After $t = 1$ there is a significant drop because the feedback mechanism being activated. After remaining low over a short time span the system appears to be heating up again. The reason for this might be that the signals used to do the feedback become unreliable when noise takes over. Yet, when a certain energy level is reached the mechanism begins to work properly again, so the kinetic energy will not exceed a certain threshold.

A picture of the "bang-bang" cooling strategy is given in figure 3.5. It shows how the trap parameter *omega* reacts to the feedback signals, which are the first and second derivative of the filtered signal. When comparing it to the actual mean position one can see that the changes happen at the proper phase of the movement. The reason why it does not have an affect on the motion is that for this plot *omega* is not fed back to the differential equations modeling the system.

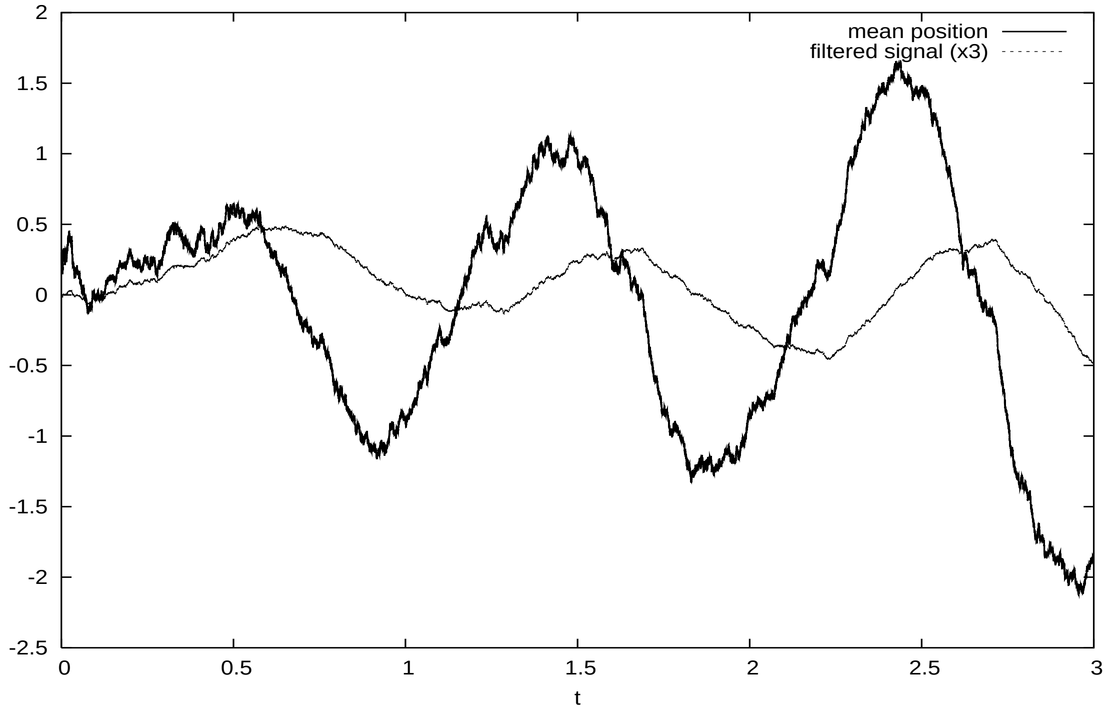

Figure 3.1: In the figure above the thick line represents the expectation value of x while the thin line represent the filter signal dy.

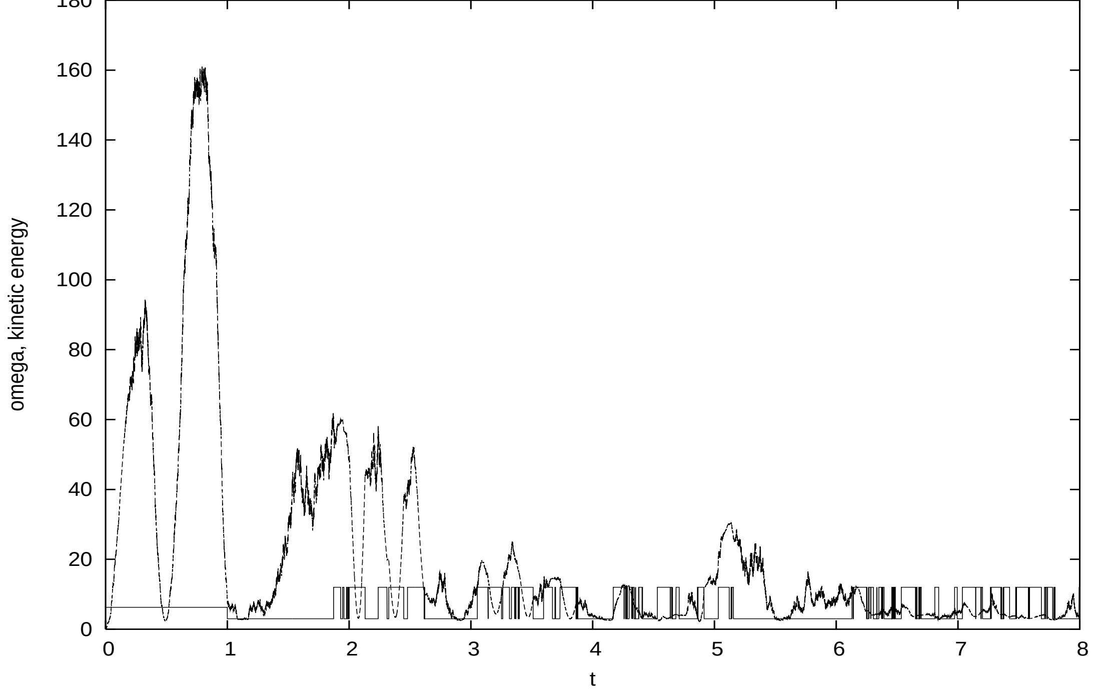

Figure 3.2: The kinetic energy changes with time. We can see the cooling effect.

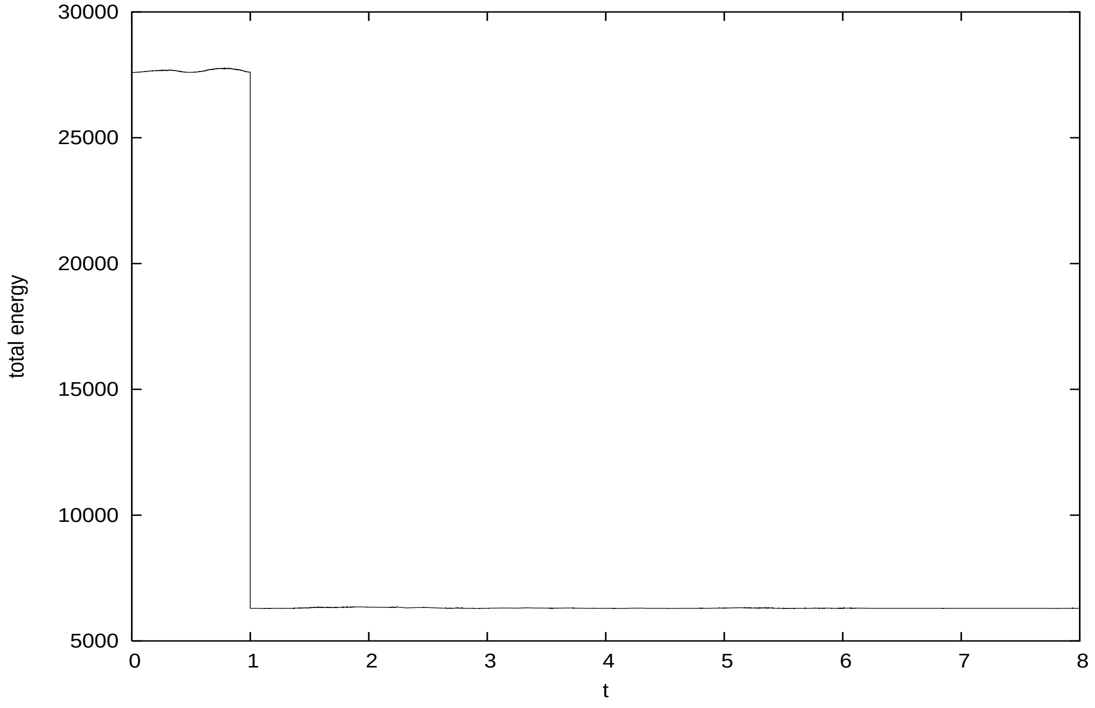

Figure 3.3: The total energy jumps when we add the feedback at $t = 1$.

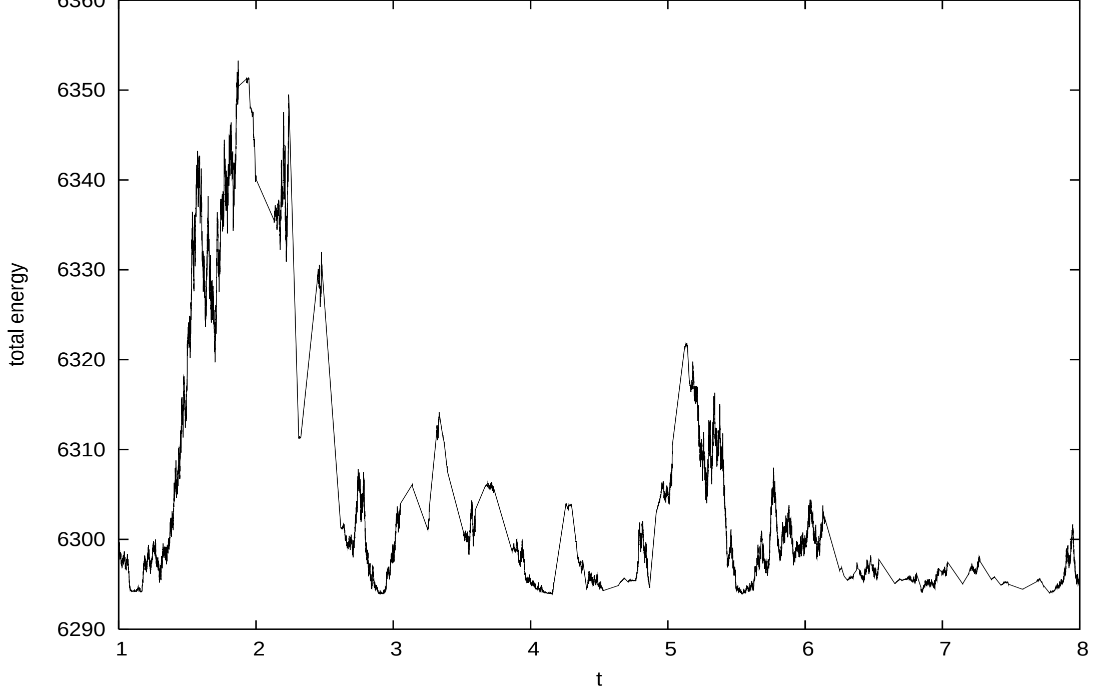

Figure 3.4: The change of total energy after $t = 1$.

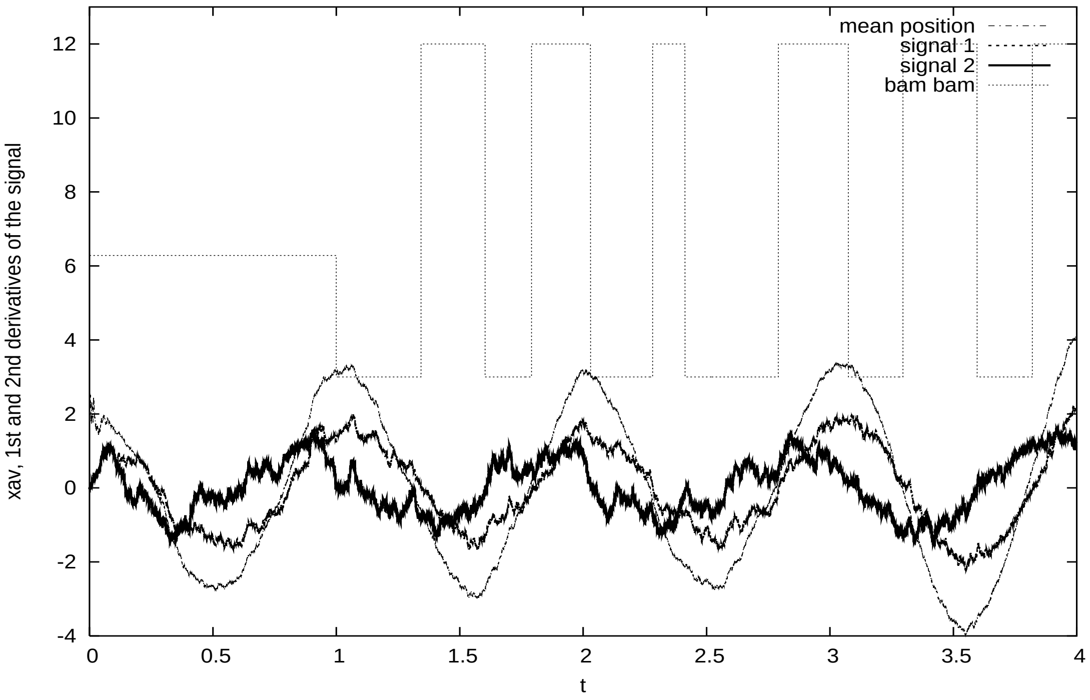

Figure 3.5: The motions of average value of position, the first and second derivatives of the filtered signal. And ω changes with time. We can see the "bang-bang".

Codes:sch1d and sderk_support

0.3 cool

```
!=========================================================================
! !  sch1d.f90 ! !  Computes evolution of Schrodinger equation in
one dimension !  by split-operator exponentials. Measurement is also
included !  in the evolution by using the stochastic equation
integrator !  modules from problem 1. The stochastic evovler takes
care !  of the position exponential part of the evolution. !  The
momentum part, including the basis transforms, is !  done by the
original evolver. ! !  To simulate a quantum feedback loop. The
measument trace !  is recorded. A low pass filter is also modeled
for the !  purpose of feedback. ! !  Ouput still consists of
distributions saved to file plus !  various parameters to stdout to
optimize feedback parameters ! !
!=========================================================================

program sch1d

  use globals
  use sderk_support
  use sderk90
  use output
  use param_parser
  use splitop_evolver
  use timing

  implicit none

  integer  :: j, m, s
  integer  :: ti, tiout
  integer, parameter :: nmaxsubsteps = 1024  ! for sderk

  real(wp) :: W
  real(wp), dimension(nmaxsubsteps) :: dWarr, I10arr


  call parse_params()

  allocate(dvarr(ndvarr))
  allocate(dv2arr(ndv2arr))

  ! Initialize sderk solver
  call init_sderk(nsubsteps)

  psi => sderk_y

  ! set initial condition (Gaussian)
  psi = exp( -(x-x0)**2/(4*sigma_x**2) ) / sqrt(sqrt(2*pi)*sigma_x)

  ! output initial condition
  call open_output_files()
  call output_stuff()
  call output_dists(first_time=.true.)

  call print_norm()
  if ( time_trace ) call print_time('s')

  W      = 0.0_wp
  deltaW = 0.0_wp
  t      = 0.0_wp


  xdist = real(psi*conjg(psi), wp)
  xav   = sum(x*xdist)*dx

  deltay = xav*dt + deltaW * sqrt(0.125_wp / k)
  V = (1-gamma*dt)*V + gamma*deltay
  Vold = V
  dvarr = 0.0_wp
  dvav = 0.0_wp
  dvavold = 0.0_wp
  dv2arr = 0.0_wp
  dv2av = 0.0_wp

  ! Main evolution loop
  j = 1
  do tiout = 1, Ntout     ! this is the loop over output times
    do ti = 1, toutevery  ! this is a loop over times between outputs

      xdist = real(psi*conjg(psi), wp)
      xav   = sum(x*xdist)*dx

      ! integration step with sderk; t is autom. updated
      call sderk(t, t + dt*0.5_wp)

      dWarr(1:nsubsteps)  = sderk_get_I1()
      deltaW = sum(dWarr(1:nsubsteps))
      I10arr(1:nsubsteps) = sderk_get_I10()
      do m = 1, nsubsteps
        W = W + dWarr(m)
      end do

      call splitop_onestep()  ! this takes care of momentum part

      xdist = real(psi*conjg(psi), wp)
      xav   = sum(x*xdist)*dx

      ! integration step with sderk; t is autom. updated
      call sderk(t, t + dt*0.5_wp)

      dWarr(1:nsubsteps)  = sderk_get_I1()
      deltaW = deltaW + sum(dWarr(1:nsubsteps))
      I10arr(1:nsubsteps) = sderk_get_I10()
      do m = 1, nsubsteps
        W = W + dWarr(m)
      end do

      xdist = real(psi*conjg(psi), wp)
      xav   = sum(x*xdist)*dx

      ! Here we calculate the measurement trace
      deltay = xav*dt + deltaW * sqrt(0.125_wp / k)

      ! This is the lowpass filter's output
      V = (1-gamma*dt)*V + gamma*deltay

      do s = 1, ndvarr-1
        dvarr(ndvarr-s+1) = dvarr(ndvarr-s)
      end do
      dvarr(1) = (V - Vold)
      Vold = V
      dvav = sum(dvarr)

      do s = 1, ndv2arr-1
        dv2arr(ndv2arr-s+1) = dv2arr(ndv2arr-s)
      end do
      dv2arr(1) = dvav - dvavold
      dvavold = dvav
      dv2av = sum(dv2arr)

      if ( (abs(dv2av) .lt. 0.0005_wp) .AND. (t .gt. 1) ) omega = minwx
      if ( (abs(dvav)  .lt. 0.0005_wp) .AND. (t .gt. 1) ) omega = maxwx

      phi = fftsf(-1,psi)*(dx/sqrt(2.0_wp*pi))
      ekin = sum(p*p*real(phi*conjg(phi),wp))*dp*0.5_wp
      epot = sum(omega*omega*x*x)*dx*0.5_wp

      write(*,format1) t, wx, xav, V, dvav, dv2av

      psi = psi/(sqrt(sum(xdist)*dx))

    end do

    ! note: for a stochastic code, you may want to make many more
    !   calls to output_stuff() than to output_dists()
    call output_stuff()       ! this writes expectation values, etc
    call output_dists()       ! this writes wave functions; *always* call
                              !   *after* output_stuff()

    if ( time_trace ) then
      write(0,*) 'Wave function computed at t = ', t
      write(0,*) '  (step ', tiout, ' of ', Ntout, ')'
    end if
    call print_norm()                  ! print wave function norm to std err
    if ( time_trace ) call print_time('m')  ! print elapsed time to std err
  end do

  ! close output files
  call cleanup()

end program sch1d
```

```
!----------------------------------------------------------------------
! !  support module for sdesample.f90 sample program to demonstrate
the !    sderk90.f90 stochastic integrator !
!-----------------------------------------------------------------------

module sderk_support

  use globals

  ! set precision of calculation to double precision
  !integer, parameter  :: sderk_prec = selected_real_kind(p=14)
  !integer, parameter  :: wp = sderk_prec

  ! use implicit order 1.5 method for integration; just leave the tolerance
  integer, parameter  :: sderk_mf = 23
  real(wp), parameter :: sderk_tol = epsilon(1.0_wp) * 100

  ! use good random number generator, just leave this
  integer, parameter  :: sderk_rand_mf = 301

  ! declare storage; you may need to change to complex or the dimension
  complex(wp), dimension(Nx), save, target :: sderk_y
  complex(wp), dimension(Nx), save ::                                    &
           sderk_a, sderk_b, sderk_c, sderk_ybar,                       &
           sderk_aybarp, sderk_aybarm, sderk_bybarp, sderk_bybarm,      &
           sderk_aphip, sderk_aphim, sderk_bphip, sderk_bphim


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! !    I am going to add xav to this module so we it can be used !
when writing the equations of motion for x !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!



contains



! Subroutines that implement equations of motion, to be called !
by sderk; note that you need to supply *two* routines, !   one to
return the dt part and one to return the dW part.

subroutine sderk_func_a(t, psi, psidot)
  implicit none
  real(wp) :: t
  complex(wp), dimension(Nx) :: psi, psidot

  ! this is the Ito form
  ! return deterministic derivative
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! !   We obviously had to change the equations of motion to the
stocahstic !   schrodinger equation which is what we want to solve.
Like before, !   Below is the deterministic part. ! !   The full
equation of motion is given in the XXXXXXXX module ! !
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

  psidot = ( (-i/hbar)*0.5_wp*omega*omega*x*x - k*(x-xav)*(x-xav) ) * psi


  return
end subroutine sderk_func_a

subroutine sderk_func_b(t, psi, psidot)
  implicit none
  real(wp) :: t
  complex(wp), dimension(Nx) :: psi, psidot

  ! return stochastic derivative
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! !   The stochastic part of the equation of motion had to be change
too !
!+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

  psidot = ( sqrt(k*2.0_wp)*(x-xav) ) * psi

  return

end subroutine sderk_func_b


end module sderk_support
```