Numerical Quantum Optics PHYS686 Project 1

Mark Rodenberger, Nima Dinyari, Xiaolu Cheng

April,2007

Problem 1

We developed a program that is used to solve the optical Bloch equations for the two-level atom.

$$\partial_t\langle\sigma_x\rangle = -\frac{1}{2}\langle\sigma_x\rangle + \Delta\langle\sigma_y\rangle$$

$$\partial_t\langle\sigma_y\rangle = -\Delta\langle\sigma_x\rangle - \frac{1}{2}\langle\sigma_y\rangle - 2\Omega\rho_{ee} + \Omega$$

$$\partial_t\rho_{ee} = \frac{\Omega}{2}\langle\sigma_y\rangle - \rho_{ee}$$

Note that we have rescaled these equations such that $t \to \Gamma t, \Delta \to \frac{\Delta}{\Gamma}$,and $\Omega \to \frac{\Omega}{\Gamma}$. Finally, we set the decay rate $\Gamma = 1$. We had a sample code to solve the equations of motion for the Harmonic Oscillator. Some changes are made in the main module and odeab_support to fit them to our case.(Codes are attached.) The program has the following usage:

```
usage: bloch <Omega> <Pe0> <tstep> <tfinal>

where

<Omega> is the (scaled) Rabi frequency

<Pe0> is the excited-state population at t=0

<tstep> is the (scaled) time step between each output

<tfinal> is the final time for the integration
```

Detunning can be ajusted in the code. The output is four columns: scaled time, $\langle\sigma_x\rangle, \langle\sigma_y\rangle, \rho_{ee}$. Data can be sent to Gnuplot, to create some nice plots of optical-Bloch-equation solutions; see below. They represent Rabi oscillations, the exponential decay of the populations and coherence between the ground and excited states, and steady-state value. We also made plots that show the difference between the numercial solution of ρ_{ee} and the analytic solution that was obtained by using Torrey's method.

It can be seen that the difference is quite small(1.5e-13), but there are also oscillations and decay. That's because when the variation of the function is larger, the numerical method is less accurate. Nevertheless, they are almost perfectly close to the analytic solutions. Here are some plots for different parameters. Note time here means real time times decay rate Γ.

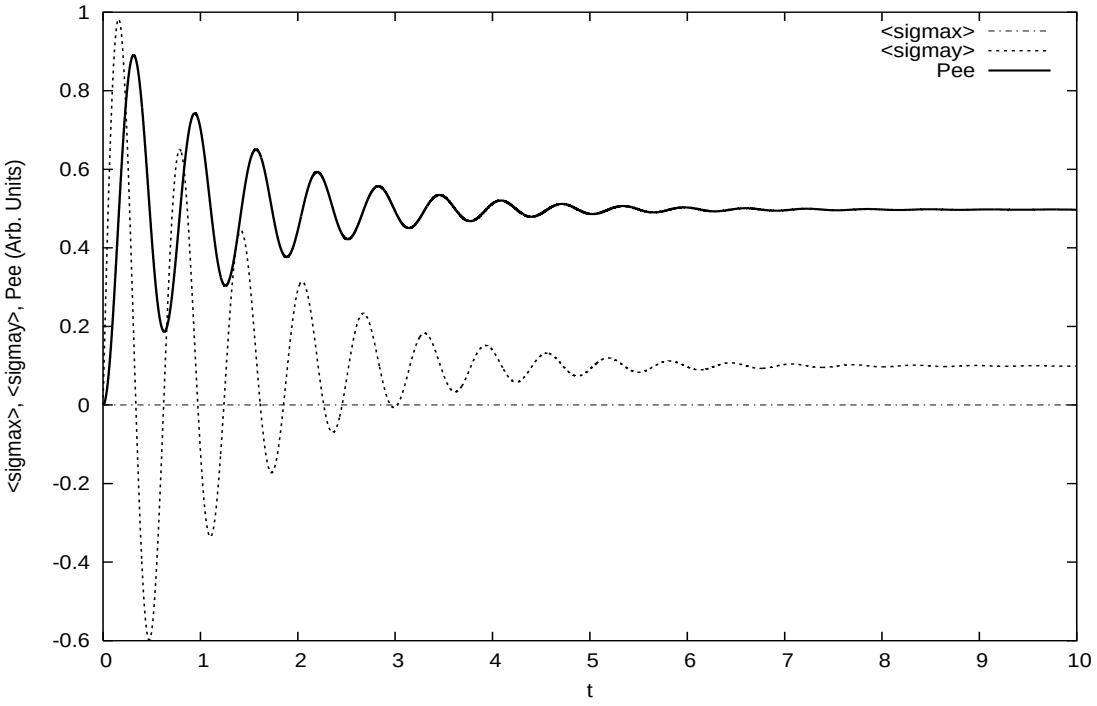

Figure 1.1: Motions of $\sigma_x, \sigma_y, \rho_{ee}.\Omega = 10\Gamma, \rho_{ee}(0) = 0, \Delta = 0$

For $\Delta = 0$, the analytical solution is

$$\rho_{ee}(t) = \frac{\Omega^2}{2\Omega^2 + \Gamma^2}\left(1 - e^{-\frac{-3t}{4}}(\cos\Omega_\Gamma t + \frac{3\Gamma}{4\Omega_\Gamma}\sin\Omega_\Gamma t)\right)$$

where $\Omega_\Gamma = \sqrt{\Omega^2 - \left(\frac{\Gamma}{4}\right)^2}$.

We used the above equation when we found the error between the numerical solution for ρ_{ee} and the analytical solution.

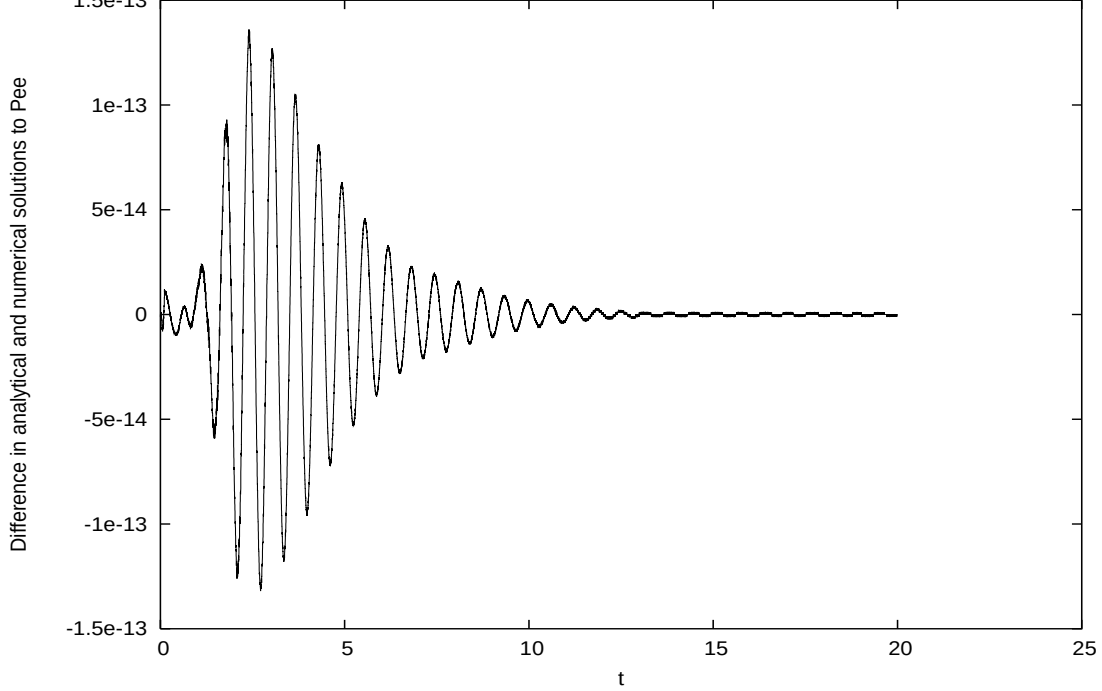

Figure 1.2: Difference in analytical and numerical solutions to $\rho_{ee}.\Omega = 10\Gamma, \rho_{ee}(0) = 0, \Delta = 0$

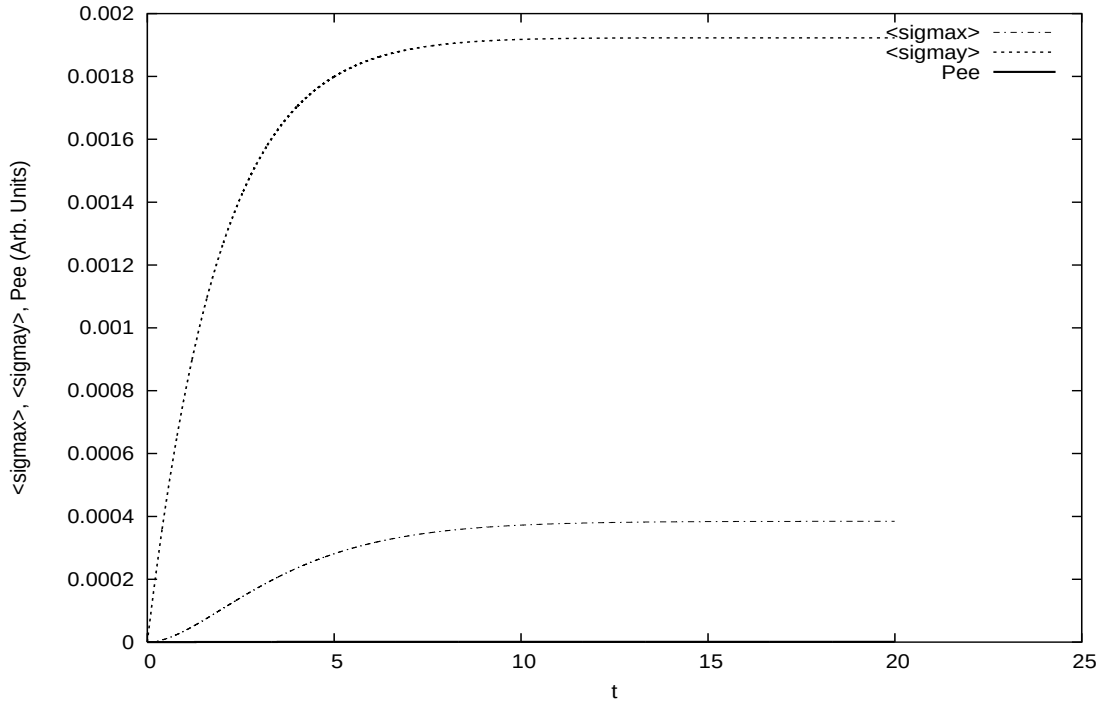

Figure 1.3: Motions of $\sigma_x, \sigma_y, \rho_{ee}.\Omega = 0.001\Gamma, \rho_{ee}(0) = 0, \Delta = 100\Omega$

For arbitrary Δ, and $\Omega \ll \Gamma$, we used the below analytic solutions

$$\rho_{ee} = \frac{\Omega^2}{\Gamma^2 + 4\Delta^2} \left(1 + d^{-\Gamma t} - 2e^{-\frac{\Gamma t}{2}} \cos \Delta t\right)$$

to compare the numerical and analytical solutions for ρ_{ee}.

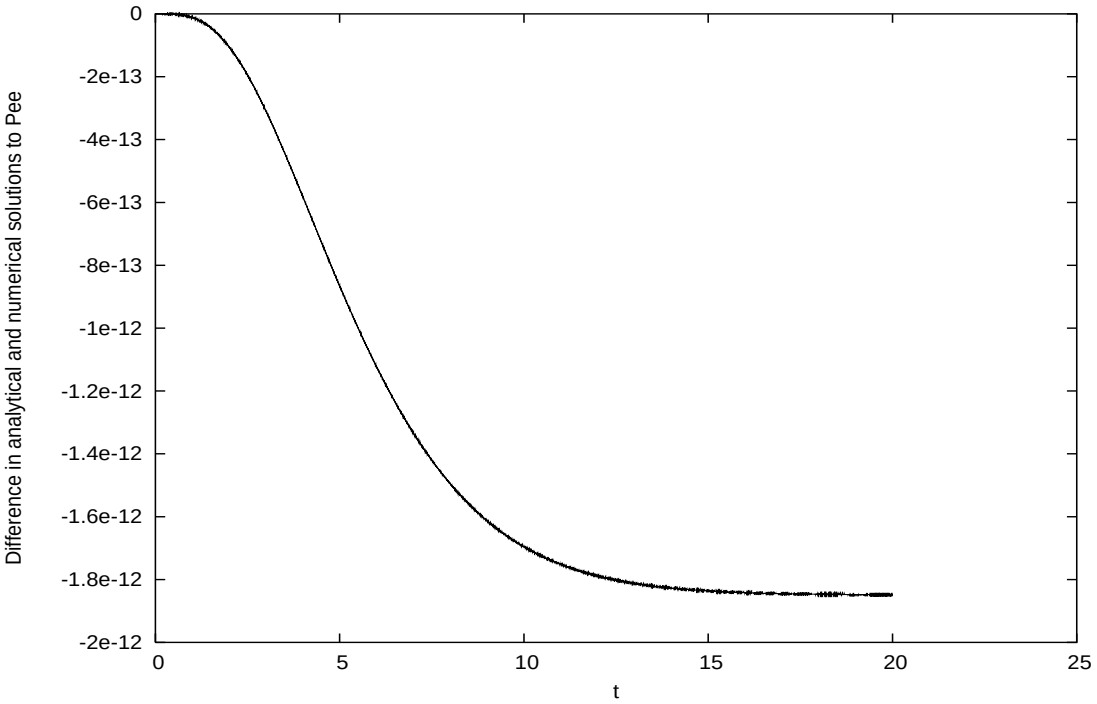

Figure 1.4: Difference in analytical and numerical solutions to ρ_{ee}.$\Omega = 0.001\Gamma, \rho_{ee}(0) = 0, \Delta = 100\Omega$

For weak excitation, oscillations disappear. The values of coherence and population of excited state are rather small. This also makes sense that atoms will stay in ground state when interaction with light is weak.

Codes: bloch and odeab_support

0.1 bloch

```
!---------------------------------------------------------------------
! !  bloch.f90 ! ! Important note: these solutions are for the
equations of motion in ! odeab_support.f90 file. We assume here that
the case at hand is for ! homogenous broadening (gamma_perp = gamma
/ 2). Also, we are using ! scaled units such that t = gamma * t,
omega = omega / gamma, and ! delta = delta / gamma. After this
rescaling gamma = 1. ! ! !  This software is to solve the optical
blcoh equations for a two !  two level system and it will give you
the numerical results for !  <simgax>, <sigmay>, and rho_ee; here,
<sigmax> and <sigmay> are !  the x and y component of the bloch
sphere and rho_ee is the excited !  state population. It works for
arbitrary detuning. However, one can only !  compare the numerical
results to the exact solutions only in two cases !  (1) when the
atom is initially in the ground state with zero coherence !  between
the ground and excited state with homogenous broadening, !  ie.
gamma_perp = gamma / 2. These exact solutions are give at !  the
bottom of this code. To reference these solutions you can look at !
Professor Daniel Steck's QOs notes from the University of Oregon. !
!  (2) The other exact solution that you can compare the numerical !
solution is the one where we have omega << gamma and gamma_perp = !
gamma / 2. ! !  Equations of motion that are being solved are: ! !
(d/dt) <sigmax> = delta * <sigmay> - 0.5 * <sigmax> !      (d/dt)
<sigmay> = -delta *<sigmax> - 2 * omega * rho_ee - 0.5<sigmay> !
+ omega !      (d/dt) rho_ee   = 0.5 * omega * <sigmax> - rho_ee ! !
!  Output is to standard output: first column is the time, !  second
is the numerical solution for sigmax, !  the third is the numerical
solution for sigmay, !  and the fourth is the numerical solution for
Pee. ! ! !  Read the extensive comments to understand the code here.
! !  This code was initially writen by Prof. Dan Steck and was
modified !  by Nima Dinyari, Mark Rodenberger, and Xiaolu Cheng to
integrate the !  above system. ! !  questions? email:
kdinyari@uoregon.edu !  last modified: April, 24th 2006
!---------------------------------------------------------------------


! comments in Fortran 90/95 start with a '!' (can be used in the
middle !   of a line)


program bloch

  ! we are using variables and subroutines in these other "modules,"
  ! so we notify the compiler to load that information
  use globals
  use odeab_support
  use odeab90
  use utilities


  ! *always* use this; it forces you to declare all your variables and
  ! prevents a lot of bugs
  implicit none


  ! declare storage for solution vector; this is specific to the odeab
  ! integrator-- only modify if you want to use real variables instead
  ! of complex (change complex -> real)
  real(wp), dimension(:), pointer :: y


  ! declare variables to use below
  ! we added the three real solutions to be
  ! used below here Pee = rho_ee above and Pee_smallomega
  ! is used for case (2) above while Pee is for zero detuning.

  !   Omega_gamma -> Rabi frequency in the presence of damping to be used
  !                  in the exact solutions in Dan's notes as part of the
  !                  the roots of f(s).
  !                  (also declared in globals.f90)
  !   Delta  -> Detuning (also declared in globals.f90)

  real(wp)      :: t, sigmax, sigmay, Pee, Pee_small_omega
  character(64) :: format1, buff
  integer       :: k, nout


  ! declare command-line arguments:
  !   Pe0    -> (real) initial amplitude of the excited state population
  !   Omega  -> Rabi frequency (declared in globals.f90)
  !   tstep  -> time step for integrator output
  !   tfinal -> final time of integrator output
  real(wp) :: Pe0, tstep, tfinal


  !!!!!! Get arguments from the command line
  !   unfortunately this was not standardized until Fortran 2003,
  !   so these commands vary among Fortran 90/95 compilers.
  !   We will use the new standard commands, which are supported
  !   by g95; they get characters, and we use 's2r' (in utilities.f90)
  !   to convert them to real numbers.
  ! Always make sure the user's input is reasonable:

  if ( command_argument_count() .ne. 4 )  call usage()
  call get_command_argument(1, buff)
  Omega = s2r(buff)                      !made the first arg omega

  call get_command_argument(2, buff)
  Pe0 = s2r(buff)                        !made the second arg Pe0

  call get_command_argument(3, buff)
  tstep = s2r(buff)
  if ( tstep .le. 0.0_wp ) then
    write(0,*)  "Error: inappropriate tstep"
    call usage()
  end if

  call get_command_argument(4, buff)
  tfinal = s2r(buff)
  if ( tfinal .le. 0.0_wp .or. tfinal .lt. tstep ) then
    write(0,*)  "Error: inappropriate tfinal"
    call usage()
  end if

  nout = nint(tfinal/tstep)
  if ( nout .gt. 50000 ) then
    write(0,*)  "Error: too many steps!"
    call usage()
  end if


  ! the integrator needs you to do this (just leave it)
  y => odeab_y


  ! Set initial values, detuning, and delta gamma
  ! note that '0.0_wp' means the number '0.0' cast into the precision 'wp'
  !   good practice so you don't introduce random errors by accidentally
  !   using single precision

  y(1) = 0.0_wp        !intially zero since no coherence intially zero as well
  y(2) = 0.0_wp        !same as above
  y(3) = Pe0 * 1.0_wp  !the user will set the initial value of Pee
  delta = omega *  100.0_wp       !sets the detuning
  t    = 0.0_wp        !start at zero time
  Omega_gamma = sqrt((Omega)**2 - 0.0625)


  ! Output initial condition
  ! format1 is a format string: there are 3 floating point numbers with
  !   spaces between, each number is formatted to use up to 21
  !   characters, with up to 15 decimal places.
  ! write(*, format1) means write to standard output ('*') using
  !   'format1' to format the output.

  !---------------------------------------------------------------------
  ! We want to added a new column to the output of our function
  ! so all we did is add a new one to the format and write
  ! the names of the functions below are given above
  !---------------------------------------------------------------------

  format1 = "(f21.15,'  ', f21.15, '  ', f21.15, '  ', f21.15)"
  write(*,format1) t, y(1), y(2), Pe0


  !!!!!! Main integration loop

  !---------------------------------------------------------------------
  ! We didnt change this
  !---------------------------------------------------------------------


  do k = 1, nout
```

```
    ! this advances the solution array 'y' from t to t+tstep
    call odeab(t, t+tstep)

    ! if any problems were encountered, this will report it
    call handle_odeab_error()

  !---------------------------------------------------------------------
  ! Below we will calculate the exact solution and output the results of
  ! numerical solution. The reason we have these here is because one might
  ! want to compare the numerical solution to the exact ones calculated in
  ! notes.
  !---------------------------------------------------------------------


    ! this is the exact solution for sigmax
    ! because you tell us that there is no coherence
    ! between the ground and excied state we set this
    ! equal to zero

    sigmax = 0

    ! this next line is the exact solution for sigmay on
    ! page 150 of the combined notes equation 5.165

    sigmay = Omega / (Omega**2 + 0.5) * ( 1 - EXP(-0.75 * t) &
            * ( cos(Omega_gamma * t) &
            - ( (Omega**2 - 0.25) / Omega_gamma ) &
            * sin(Omega_gamma * t) ) )

    ! this is the exact solution for Pee which is given by
    ! using the exact solution for the sigmaz on page
    ! 151 equation 5.170 and using the
    ! fact that sigmaz = 2 * Pee - 1

    Pee = ( (((Omega**2) * 0.5)/(Omega**2 + 0.5)) * ( 1 - EXP(-0.75 * t) &
         * (cos(Omega_gamma * t) &
         + (0.75 / Omega_gamma) * sin(Omega_gamma * t) ) ) )

   ! this will be the exact solution for Pee for arbitrary detuning
   ! when omega << gamma and homogenous broadening.

   Pee_small_omega = ( Omega**2 * 0.5 ) / ( 0.5 + 2 * delta**2) &
         * ( 1 + EXP(-t) - 2 * EXP(-0.5 * t) * cos(delta * t))


    write(*,format1) t, y(1) * 1.0_wp, y(2) * 1.0_wp, y(3) * 1.0_wp

  end do

  ! print performance statistics, if you're curious, to standard error
  call print_odeab_stats(cumulative = .true.)


  contains

  subroutine usage()
    ! write (0,*) means write to standard error (0), using default
    !   formatting (*)
    write(0,*) ''
    write(0,*) ''
    write(0,*) 'Usage: ./bloch <Omega> <Pe0> <tstep> <tfinal>'
    write(0,*) ' Omega  -> Rabi frequency / gamma'
    write(0,*) ' Pe0  -> initial excited state population'
    write(0,*) ' tstep  -> time step for integrator output'
    write(0,*) ' tfinal -> final time of integrator output'
    write(0,*) ''
    write(0,*) 'Output is four columns of text to standard output:'
    write(0,*) '  gamma * time, sigmax (numerical), sigmay (numerical), and Pee (numerical)'
    write(0,*) ''
    write(0,*) ''
    stop
  end subroutine usage

end program bloch


!---------------------------------------------------------------------
! !  Support module for bloch.f90  program ! !  This module contains
certain setup stuff for the integrator, !    as well as the
equations of motion to solve. The equations !    of motion are
described in bloch.f90 !
!---------------------------------------------------------------------
```

```
module odeab_support

  use globals

  ! declare precision, max # of steps, and error tolerances for integration
  integer, parameter  :: odeab_prec = wp
  integer, parameter  :: odeab_maxstp = 500
  real(wp), parameter :: odeab_atol = 1.e-13_wp
  real(wp), parameter :: odeab_rtol = 1.e-13_wp

! the following lines declare internal storage for the odeab
  ! integrator; just leave it, except:
  !   1. change the dimension to match the number of integration variables
  !   2. if your variables are all real, change "complex"es to "real"
  type odeab_type
    real(wp), dimension(3) :: phi
  end type
  type(odeab_type), dimension(16), save :: odeab_idx
  real(wp), dimension(3), target :: odeab_y
  real(wp), dimension(3), save   :: odeab_yy, odeab_p, odeab_yp
  real(wp), dimension(3), save   :: odeab_wt

  ! declare basic management stuff for integrator (just leave it)
  integer :: odeab_istate = 1
  logical, parameter :: odeab_stop = .false.


contains

! Subroutine that implements equations of motion, to be called !
by odeab; implements simple complex rotation

subroutine odeab_func(t, y, ydot)
  implicit none
& real(wp), intent(in) :: t
  real(wp), dimension(3), intent(in)  :: y
  real(wp), dimension(3), intent(out) :: ydot

  ! return derivative
  ! Below we will use the equations of motion on page 145 of the combined
  ! notes equations 5.119. So here we represent y(1) sigmax, y(2) sigmay,
  ! and y(3) is Pee. Omega and delta will declared in globals.f90
  !


  ydot(1) = delta * y(2) - 0.5 * y(1)
  ydot(2) = -delta * y(1) - 0.5 * y(2) - 2 * omega * y(3) + omega
  ydot(3) = 0.5 * omega * y(2) - y(3)

  return
end subroutine odeab_func

end module odeab_support
```

Problem 2

This section deals with numerically solving the two level atom with stochastic emission events. First, we know that the two level atom can be described by the unconditioned master equation.

$$\partial_t \rho = -\frac{i}{\hbar}[H_A + H_{AF}, \rho] - \frac{\Gamma}{2}[\sigma^\dagger \sigma, \rho]_+ + \Gamma \sigma \rho \sigma^\dagger . (1)$$

The stochastic master equation (sme) incorporates the random photon detection event into the model.

$$d\rho = -\frac{i}{\hbar}[H_A + H_{AF}, \rho]dt - \frac{\Gamma}{2}[\sigma^\dagger \sigma, \rho]_+ dt + \Gamma \langle \sigma^\dagger \sigma \rangle \rho dt + \left(\frac{\sigma \rho \sigma^\dagger}{\langle \sigma^\dagger \sigma \rangle} - \rho \right) dN . (2)$$

We can show that taking ensemble average leads back to the unconditioned equation.

$$\langle \langle dN \rangle \rangle = \Gamma \langle \sigma^\dagger \sigma \rangle dt$$

$$d\rho = -\frac{i}{\hbar}[H_A + H_{AF}, \rho]dt - \frac{\Gamma}{2}[\sigma^\dagger \sigma, \rho]_+ dt + \Gamma \langle \sigma^\dagger \sigma \rangle \rho dt + \left(\frac{\sigma \rho \sigma^\dagger}{\langle \sigma^\dagger \sigma \rangle} - \rho \right) \Gamma \langle \sigma^\dagger \sigma \rangle dt$$

$$d\rho = -\frac{i}{\hbar}[H_A + H_{AF}, \rho]dt - \frac{\Gamma}{2}[\sigma^\dagger \sigma, \rho]_+ dt + \Gamma \sigma \rho \sigma^\dagger dt$$

That agrees with

$$\partial_t \rho = -\frac{i}{\hbar}[H_A + H_{AF}, \rho] - \frac{\Gamma}{2}[\sigma^\dagger \sigma, \rho]_+ + \Gamma \sigma \rho \sigma^\dagger$$

Surprisingly there is a so called stochastic Schrödinger equation (sse) which is equivalent to the sme. It can be used to simplify the problem.

$$d|\psi\rangle = -\frac{i}{\hbar}(H_A + H_{AF})|\psi\rangle dt + \frac{\Gamma}{2}(\langle \sigma^\dagger \sigma \rangle - \sigma^\dagger \sigma)|\psi\rangle dt + \left(\frac{\sigma}{\sqrt{\langle \sigma^\dagger \sigma \rangle}} - 1 \right) |\psi\rangle dN . (3)$$

To show that the stochastic Schröinger equation and the stochastic master equation are equivalent one can take the differential of the pure state density operator. It has to be taken into account that the second order terms do not all vanish due to $(dN)^2$.

Density matrix

$$\rho = |\psi\rangle\langle\psi|$$

$$d\rho = (d|\psi\rangle)\langle\psi| + |\psi\rangle d(\langle\psi|) + d(|\psi\rangle)d(\langle\psi|)$$

Conjugate of (3) is

$$d\langle\psi| = \frac{i}{\hbar}\langle\psi|(H_A + H_{AF})dt + \frac{\Gamma}{2}\langle\psi|(\langle\sigma^\dagger\sigma\rangle - \sigma^\dagger\sigma)dt + \langle\psi| \left(\frac{\sigma^\dagger}{\sqrt{\langle\sigma^\dagger\sigma\rangle}} - 1 \right) dN$$

then

$$d\rho = -\frac{i}{\hbar}(H_A + H_{AF})\rho dt + \frac{\Gamma}{2}(\langle\sigma^\dagger\sigma\rangle - \sigma^\dagger\sigma)\rho dt + \left(\frac{\sigma}{\sqrt{\langle\sigma^\dagger\sigma\rangle}} - 1\right)\rho dN$$

$$+\frac{i}{\hbar}\rho(H_A + H_{AF})dt + \frac{\Gamma}{2}\rho(\langle\sigma^\dagger\sigma\rangle - \sigma^\dagger\sigma)dt + \rho\left(\frac{\sigma^\dagger}{\sqrt{\langle\sigma^\dagger\sigma\rangle}} - 1\right)dN$$

$$+\emptyset\left((dt)^2\right) + \left(\frac{\sigma}{\sqrt{\langle\sigma^\dagger\sigma\rangle}} - 1\right)\rho\left(\frac{\sigma^\dagger}{\sqrt{\langle\sigma^\dagger\sigma\rangle}} - 1\right)(dN)^2$$

$$d\rho = -\frac{i}{\hbar}[H_A + H_{AF}, \rho]dt + \Gamma\langle\sigma^\dagger\sigma\rangle\rho dt - \frac{\Gamma}{2}[\sigma^\dagger\sigma, \rho]_+ dt - \rho dN + \frac{\sigma\rho\sigma^\dagger}{\langle\sigma^\dagger\sigma\rangle}dN$$

is equation (2), and is equivalent to (1)

Now that the equivalence of these equations has been shown we can go about to derive the proper dynamical equations for the the state vector coefficients C_g and C_e.

$$dC_e = (i\Delta C_e - i\frac{\Omega}{2}C_g + \frac{\Gamma}{2}|C_e|^2 C_e - \frac{\Gamma}{2}C_e)dt - C_e dN$$

$$dC_g = (-i\frac{\Omega}{2}C_e + \frac{\Gamma}{2}C_g|C_e|^2)dt + \left(\frac{C_e}{|C_e|} - C_g\right)dN$$

One can transform to scaled units (time multiplied by decay rate) where the constants must be divided by the decay rate. An appropriate redefinition of these constants makes it possible to remove the decay rate from the equations. One must only remember to divide the time coordinate by it to obtain the actual time in seconds.

The system is described by two first order differential equations for complex functions. This made it possible to adapt the given sample code (from the file 'randsample.tgz'), which solves the harmonic oscillator with random resets, and apply it to the gives sse. The number of equations was the same so only their form had to be changed in the file 'odeab_sample.f90'. There were two significant changes that had do be done on the main file, which was renamed from 'hoscr.f90' to 'sse.f90' (the makefile was manipulated appropriately). The first was to change the reset condition in the integration loop. In the code's current form a reset is supposed to occur when the atom's decay probability in a certain small time span is greater or equal to a uniformly distributed number between 0 and 1. The second was to not save the solutions to the array directly but to compute the desired values from the evolving coefficients and save those to the storage array. Format and storage array dimension had to be adapted. The program now needs scaled rabi frequency, scaled integrator time step, scaled final time and the number of single trajectories to be averaged over as input. The random generator seed is an optional input. Detuning is set in the code itself. The output goes to standard out and consists of four columns: scaled time, $\langle\sigma_x\rangle, \langle\sigma_y\rangle$ and the excited state population. To test the programs functionality the executable was fed with different parameters which produced the following plots. (Figure 2.1 and 2.2)

To show the presence of the statistical jumps here is a plot of single trajectories for the excited state population (each with a different seed number). Notice all trajectories starting out together but after some stochastical jumps they dephase. This would produce a damping effect if averaged over. (Figure 2.3)

The damping effect can be seen in the next plot showing several numerical solutions for the excited state population each with a different number of trajectories to be averaged over compared to the analytical solution of the damped two-level atom in the appropriate regime. A higher number of trajectories brings the graph closer to the exact solution. 10000 is already very close.(Figure 2.4)

Since the decay probability used in the program is only correct for infinitely small time steps one can examine which step size is needed to make numerical and exact solutions match. The following plot shows numerically calculated excited state populations with different time step sizes and the exact solution. A 0.01 scaled step shows good (plain eye) convergence. (Figure 2.5)

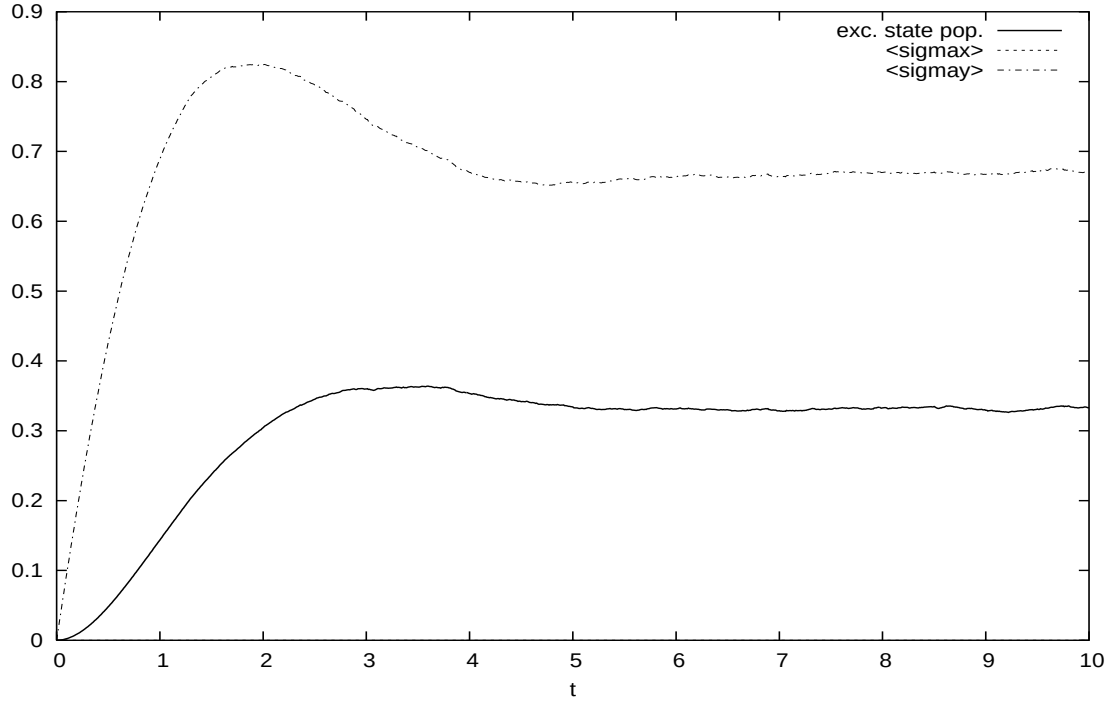

Figure 2.1: general plot of numerical solutions $\Omega = \Gamma, tstep = 0.01, ntraj = 10000$

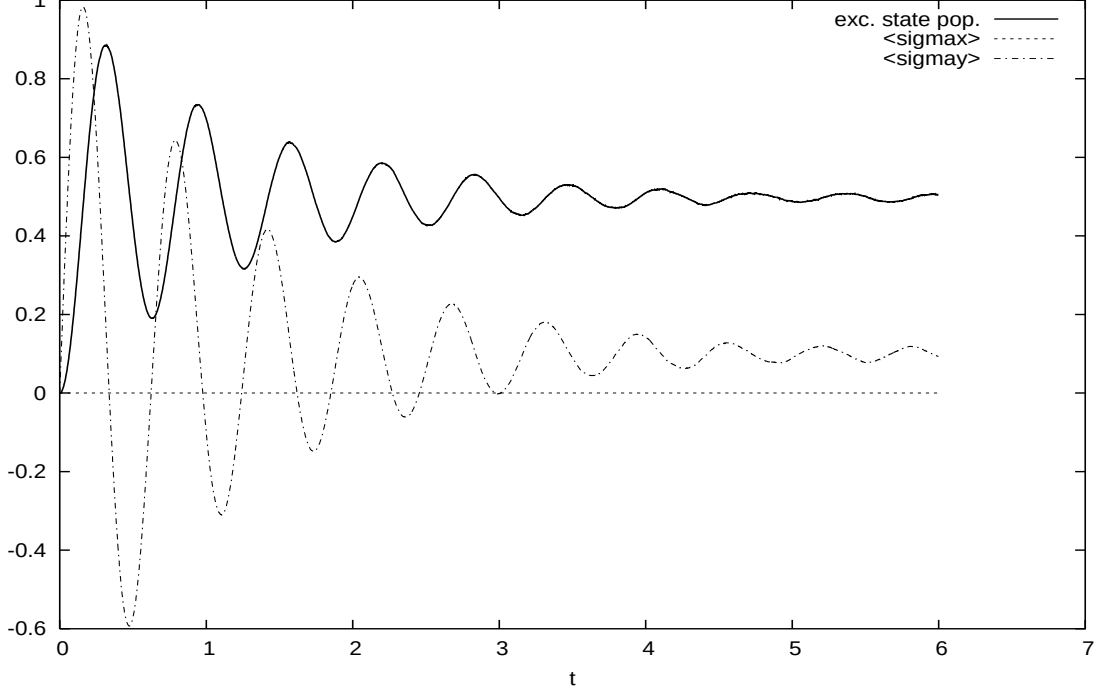

Figure 2.2: general plot of numerical solutions $\Omega = 10\Gamma, tstep = 0.01, ntraj = 10000$

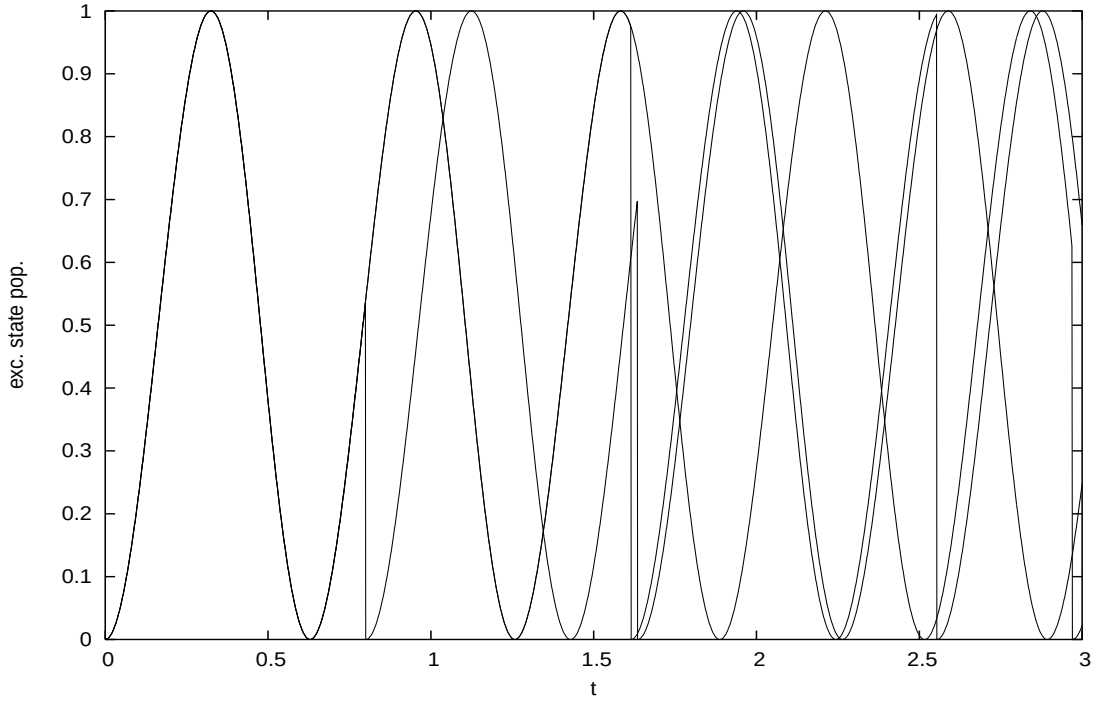

Figure 2.3: plot of three single trajectories for excited state population showing the dephasing due to emission $\Omega = 10\Gamma, tstep = 0.001, seed1, 3 and 4$

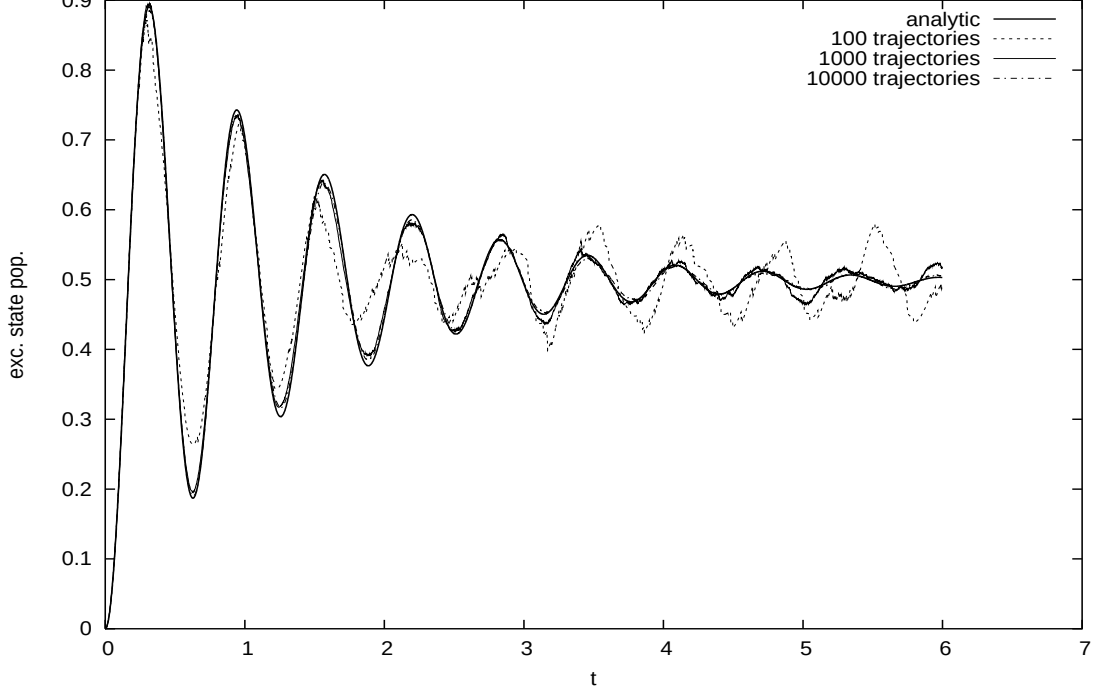

Figure 2.4: comparison of several numerical solutions for excited state population with different ntraj to analytic solution $\Omega = 10\Gamma, tstep = 0.01$

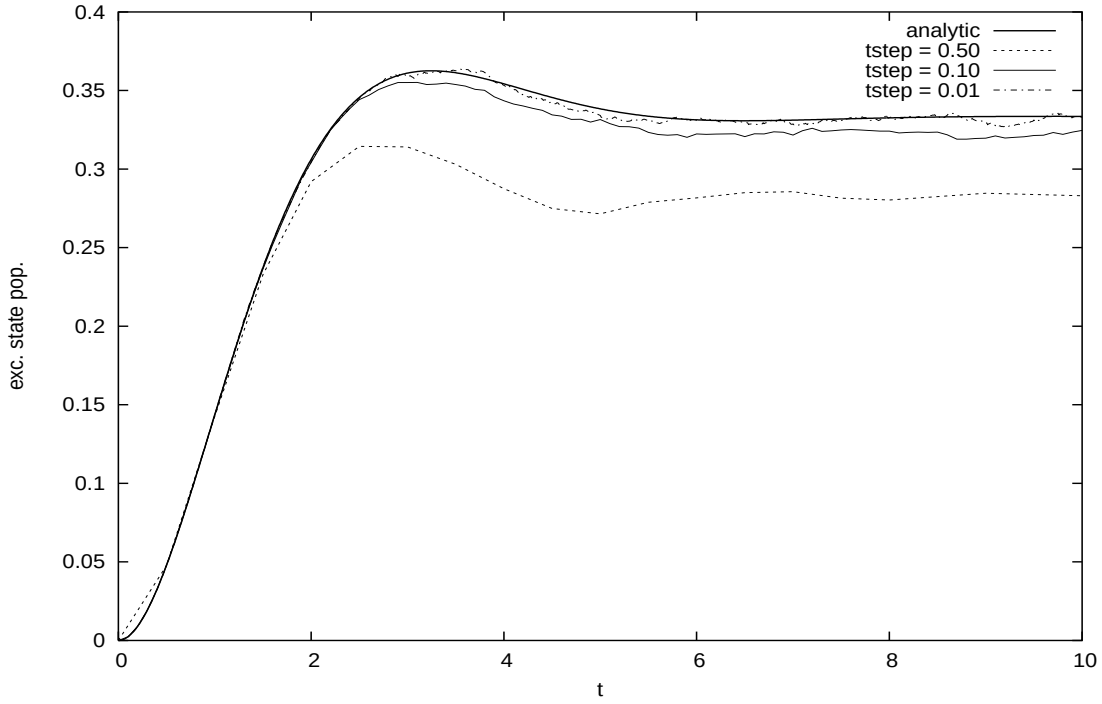

Figure 2.5: comparison of several numerical solutions for excited state population with different time step size to analytic solution $\Omega = \Gamma, ntraj = 10000$

Codes: sse and odeab_support

0.2 sse

```
!-----------------------------------------------------------------------
! !  sse.f90 ! !  Code for solving a simple differential equation !
using "canned" integrator odeab.  The problem is !  the stochastic
schrdinger eq. for the 2-level atom: ! !      d(|psi>) = -(i/h) H
|psi> dt + G/2 (<o*o>-o*o) |psi> dt !                + (
(o/sqrt(<o*o>))-1 ) |psi> dN ! !      o = sigma, o* = sigma dagger,
G = gamma ! !  The atomic state has the following form: ! ! |psi> =
cg |g> + ce |e> ! !  Coefficients' evolution is determined by
schrdinger eq. !  The code aims at integrating the implied
coefficient odes. ! !  Output is to standard output: first column is
the time, !  second, third and fourth are the numerical solutions
for !  sigmax, sigmay and rhoee. ! !  This code is taken from D.
Steck's example code !  for solving the harmonic oscillator with
stochastic !  reset and altered by Nima Dinyari, Xiaolu Cheng and !
Mark Rodenberger to solve the two level atom. !
!-----------------------------------------------------------------------



program sse

  ! we are using variables and subroutines in these other "modules,"
  ! so we notify the compiler to load that information
  use globals
  use odeab_support
  use odeab90
  use random_pl
  use utilities


  implicit none



  ! declare storage for solution vector; this is specific to the odeab
  ! integrator-- only modify if you want to use real variables instead
  ! of complex (change complex -> real)
  complex(wp), dimension(:), pointer :: y


  ! declare variables to use below
  real(wp)      :: t
  character(64) :: format1, buff
  integer       :: k, nout, traj


  ! declare dynamic storage array
  complex(wp), dimension(:,:), allocatable :: yout

  !declare memory variable to calculate emission probability
  complex(wp) :: y2old

  ! declare command-line arguments:
  !
  !   Omega  -> rabi frequency (declared in globals.f90)
  !   tstep  -> time step for integrator output
  !   tfinal -> final time of integrator output
  !   ntraj  -> number of trajectories to average together
  !   seed   -> number to seed the random number generator
  !             (same seed gives same sequence of "random" numbers)
  !             this argument is optional
  !-----------------------------------------------------------------------


  ! Below we edited the file because the usage only calls for a minum of
  ! four arguments and one optinal one. So command_argument_cound .ne. 5
  ! goes to four and command_argument_count 6 goes to 5. Then we erased the
  ! the y0 line and promoted all the others #s up in the call ....(#, buff)
  ! to make sure it was stored in the right place.
  !-----------------------------------------------------------------------




  real(wp) :: tstep, tfinal, omega_gamma
  integer  :: ntraj
  integer(rpk) :: seed  ! need to be special integer type


  !!!!!! Get arguments from the command line
  ! unfortunately this was not standardized until Fortran 2003,
  !   so these commands vary among Fortran 90/95 compilers.
  !   We will use the new standard commands, which are supported
  !   by g95; they get characters, and we use 's2r' (in utilities.f90)
  !   to convert them to real numbers.
  ! Always make sure the user's input is reasonable:
  if ( command_argument_count() .ne. 4 .and. command_argument_count() .ne. 5 ) &
     call usage()

  call get_command_argument(1, buff)
  Omega = s2r(buff)

  call get_command_argument(2, buff)
  tstep = s2r(buff)
  if ( tstep .le. 0.0_wp ) then
    write(0,*)  "Error: inappropriate tstep"
    call usage()
  end if

  call get_command_argument(3, buff)
  tfinal = s2r(buff)
  if ( tfinal .le. 0.0_wp .or. tfinal .lt. tstep ) then
    write(0,*)  "Error: inappropriate tfinal"
    call usage()
  end if

  nout = nint(tfinal/tstep)
  if ( nout .gt. 5000000 ) then
    write(0,*)  "Error: too many steps!"
    call usage()
  end if

  call get_command_argument(4, buff)
  ntraj = s2i(buff)
  if ( ntraj .lt. 1 ) then
    write(0,*)  "Error: ntraj must be a positive integer"
    call usage()
  end if
  if ( ntraj .gt. 10000000 ) then
    write(0,*)  "Error: that's a crazy number of trajectories!"
    call usage()
  end if

  ! get random number seed, if specified
  if ( command_argument_count() .eq. 5 ) then
    call get_command_argument(5, buff)
    seed = s2i(buff)
    call init_rand_pl(seed1=seed)
  end if

  ! allocate output storage array (long dimension always goes first!)
  ! second index set to 3 (this one cost some time to find; important!!)
  allocate( yout(nout+1, 3) )
  yout = 0


  ! the integrator needs you to do this (just leave it)
  y => odeab_y

  ! Set format of output
  format1 = "(f21.15,'  ', f21.15,'  ',f21.15,'  ',f21.15)"

 ! set detuning
  delta = omega*0.0_wp
  !omega_gamma = SQRT( (omega^2 - (1/4)^2))
  omega_gamma = SQRT(omega**2 - 0.0625_wp)

  !!!!!!  Main loop over trajectories
  do traj = 1, ntraj

    ! Set initial values; y(1)=cg, y(2)=ce, initially in
    !                                       ground state
    y(1) = 1.0_wp
    y(2) = 0.0_wp
    y2old = y(2)
    t    = 0.0_wp

    ! Reset the integrator
    odeab_istate = 1

    ! Save initial condition
    yout(1, 1) = yout(1, 1) + real(y(2)*conjg(y(1))+y(1)*conjg(y(2)),wp)
    yout(1, 2) = yout(1, 2) + real(i*(y(2)*conjg(y(1))-y(1)*conjg(y(2))),wp)
    yout(1, 3) = yout(1, 3) + real(y(2)*conjg(y(2)),wp)
```

```
    !!!!!! Main integration loop
    do k = 1, nout

      ! this advances the solution array 'y' from t to t+tstep
      call odeab(t, t+tstep)

      ! if any problems were encountered, this will report it
      call handle_odeab_error()

      ! reset the trajectory with decay probability
      !(tstep must be small for good accuracy!)
      if ( rand_pl() .le. real(tstep*( y2old*conjg(y2old)&
          &+y(2)*conjg(y(2)))*0.5_wp,wp) ) then
          y(1) = 1.0_wp
          y(2) = 0.0_wp
          odeab_istate = 1
          ! tell the integrator to restart the solution
      end if

      ! remember ce
      y2old = y(2)

      ! add results to storage array
      ! yout(k,1) = sigmax, yout(k,2) = sigmay, yout(k,3) = rhoee
      yout(k+1, 1) = yout(k+1, 1) + real(y(2)*conjg(y(1)),wp)
      yout(k+1, 2) = yout(k+1, 2) + real(i*(y(2)*conjg(y(1))&
                    &-y(1)*conjg(y(2))),wp)
      yout(k+1, 3) = yout(k+1, 3) + real(y(2)*conjg(y(2)),wp)

    end do  !!! main integration loop

  end do  !!! main trajectory loop


  ! divide sum by N to get average
  yout = yout / ntraj

  ! This part is for exact solution and blanked out here
  ! calculate and save exact solution
  !t = 0.0_wp
  !do k = 1, nout+1
  !  yout(k, 4) = ( (((Omega**2) * 0.5)/(Omega**2 + 0.5)) * &
  !                 &( 1 - DEXP(-0.75 * t) &
  !       * (cos(Omega_gamma * t) &
  !       + (0.75 / Omega_gamma) * sin(Omega_gamma * t) ) ) )
  !  t = t + tstep
  !end do

  ! write out results
  t = 0.0_wp
  do k = 1, nout+1
    write(*,format1) t, real(yout(k,1),wp), &
          & real(yout(k,2),wp), real(yout(k,3),wp)
    t = t + tstep
  end do



  ! print performance statistics, if you're curious, to standard error
  call print_odeab_stats(cumulative = .true.)



  contains

  subroutine usage()
    ! write (0,*) means write to standard error (0), using default
    !   formatting (*)
    write(0,*) ''
    write(0,*) ''
    write(0,*) 'Usage: sse <omega> <tstep> <tfinal> <ntraj> [<seed>]'
    write(0,*) ' omega  -> scaled rabi-frequency'
    write(0,*) ' tstep  -> scaled time step for integrator output'
    write(0,*) ' tfinal -> scaled final time of integrator output'
    write(0,*) ' ntraj  -> number of stochastic trajectories to average'
    write(0,*) ' seed   -> random number seed (optional)'
    write(0,*) ''
    write(0,*) 'Output is four columns of text to standard output:'
    write(0,*) '  time, sigmax (num), sigmay (num), excited state pop (num)'
    write(0,*) ''
    write(0,*) ''
    stop
  end subroutine usage

end program sse
```

```
!-----------------------------------------------------------------------
! !  Support module for hosc.f90 sample program to demonstrate the !
odeab integrator ! !  This module contains certain setup stuff for
the integrator, !    as well as the equations of motion to solve. !
!-----------------------------------------------------------------------

module odeab_support

  use globals

  ! declare precision, max # of steps, and error tolerances for integration
  integer, parameter  :: odeab_prec = wp
  integer, parameter  :: odeab_maxstp = 500
  real(wp), parameter :: odeab_atol = 1.e-13_wp
  real(wp), parameter :: odeab_rtol = 1.e-13_wp

  ! the following lines declare internal storage for the odeab
  ! integrator; just leave it, except:
  !   1. change the dimension to match the number of integration variables
  !   2. if your variables are all real, change "complex"es to "real"
  type odeab_type
    complex(wp), dimension(2) :: phi
  end type
  type(odeab_type), dimension(16), save :: odeab_idx
  complex(wp), dimension(2), target :: odeab_y
  complex(wp), dimension(2), save   :: odeab_yy, odeab_p, odeab_yp
  real(wp), dimension(2), save      :: odeab_wt

  ! declare basic management stuff for integrator (just leave it)
  integer :: odeab_istate = 1
  logical, parameter :: odeab_stop = .false.


contains


! Subroutine that implements equations of motion, to be called !
by odeab; implements two level atom evolution

subroutine odeab_func(t, y, ydot)
  implicit none
  real(wp), intent(in) :: t
  complex(wp), dimension(2), intent(in)  :: y
  complex(wp), dimension(2), intent(out) :: ydot

  ! return derivative
  ydot(1) = -i*0.5*omega*y(2)+0.5*y(1)*y(2)*conjg(y(2))                ! cgdot
  ydot(2) = -i*0.5*omega*y(1)+(i*delta+0.5*(y(2)*conjg(y(2))-1))*y(2) ! cedot

  return
end subroutine odeab_func

end module odeab_support
```

Problem 3

We are trying to write a program to generate quantum trjectories and investigate the quantum beats for vee atom. The possibility of quantum beats in resonance fluorescence is one of the most significant differences between lambda system and vee system. Assume the radiation from the two transitions is indistinguishable. The radiated field intensity scales as

$$\langle E^{(-)}E^{(+)}\rangle \propto \langle d^{(-)}d^{(+)}\rangle \propto \langle(\sigma_1^\dagger+\sigma_2^\dagger)(\sigma_1+\sigma_2)\rangle = \rho_{e_1e_2}+\rho_{e_1e_2}+\rho_{e_1e_2}+\rho_{e_2e_1}$$

We see that besides the sum of the excited-state populations, there are also the last two coherence terms represent interference between the two populations. In the case where the two excited states have different energies, their coherences rotate at the splitting frequency, thus leading to the quantum beats in the resonance fluorescence. In the case where the excited states are nondegenerate with splitting δ but both coupled from the ground state by the same field, we can observe steady quantum beats.

Unconditioned master equation for indistinguishable transitions

$$\partial_t\rho = -\frac{i}{\hbar}\left[H_A+H_{AF},\rho\right]+D\left[\sqrt{\Gamma_1}\sigma_1+\sqrt{\Gamma_2}\sigma_2\right]\rho$$

Counting in the effect of random photon detection, we should use the stochastic schrödinger equation

Note

$$c = \sqrt{\Gamma_1}\sigma_1+\sqrt{\Gamma_2}\sigma_2$$

$$H_A = -\hbar\Delta_1|e_1\rangle\langle e_1|-\hbar\Delta_2|e_2\rangle\langle e_2|$$

$$H_{AF} = \frac{\hbar\Omega}{2}(\sigma_1^\dagger+\sigma_1+\sigma_2^\dagger+\sigma_2) = \frac{\hbar\Omega}{2}(|e_1\rangle\langle g|+|g\rangle\langle e_1|+|e_2\rangle\langle g|+|g\rangle\langle e_2|)$$

Stochastic Schrödinger equation

$$d|\psi\rangle = -\frac{i}{\hbar}(H_A+H_{AF})|\psi\rangle dt+\frac{1}{2}(\langle c^\dagger c\rangle-c^\dagger c)|\psi\rangle dt+\left(\frac{c}{\sqrt{\langle c^\dagger c\rangle}}-1\right)|\psi\rangle dN$$

Amplitudes

$$dC_{e_1} = (i\Delta_1C_{e_1}-i\frac{\Omega}{2}C_g)dt+\frac{1}{2}(\Gamma_1|C_{e_1}|^2+\Gamma_2|C_{e_2}|^2+\sqrt{\Gamma_1\Gamma_2}C_{e_1}C_{e_2}^*+\sqrt{\Gamma_1\Gamma_2}C_{e_1}^*C_{e_2})C_{e_1}dt$$
$$-\frac{1}{2}(\Gamma_1C_{e_1}+\sqrt{\Gamma_1\Gamma_2}C_{e_2})dt-C_{e_1}dN$$
$$dC_{e_2} = (i\Delta_2C_{e_2}-i\frac{\Omega}{2}C_g)dt+\frac{1}{2}(\Gamma_1|C_{e_1}|^2+\Gamma_2|C_{e_2}|^2+\sqrt{\Gamma_1\Gamma_2}C_{e_1}C_{e_2}^*+\sqrt{\Gamma_1\Gamma_2}C_{e_1}^*C_{e_2})C_{e_2}dt$$
$$-\frac{1}{2}(\sqrt{\Gamma_1\Gamma_2}C_{e_1}+\Gamma_2C_{e_2})dt-C_{e_2}dN$$

From this, the motion for the amplitudes will be

$$dC_g = -i\frac{\Omega}{2}(C_{e_1} + C_{e_2})dt + \frac{1}{2}(\Gamma_1|C_{e_1}|^2 + \Gamma_2|C_{e_2}|^2 + \sqrt{\Gamma_1\Gamma_2}C_{e_1}C_{e_2}^* + \sqrt{\Gamma_1\Gamma_2}C_{e_1}^*C_{e_2})C_g dt$$

$$+ \left(\frac{\sqrt{\Gamma_1}C_{e_1} + \sqrt{\Gamma_2}C_{e_2}}{\sqrt{\Gamma_1|C_{e_1}|^2 + \Gamma_2|C_{e_2}|^2 + \sqrt{\Gamma_1\Gamma_2}C_{e_1}C_{e_2}^* + \sqrt{\Gamma_1\Gamma_2}C_{e_1}^*C_{e_2}}} - C_g \right) dN$$

What we want to measure is the absorption rate

$$R_{abs} = \Omega \cdot \mathrm{Im}[C_g C_{e_1}^* + C_g C_{e_2}^*]$$

Our program has the same usage as in the second problem. And we set $\Gamma_1 = 1$, and Γ_2 can be adjusted. According to the output of our program, typically, with time increasing, the absorption is oscillating. There is some negative absorption. The explanation for this is that there is emission at that time. As a function of splitting, there are two peaks and zero absorption when splitting is zero. We also make 3-D plots to show the absorption changes with time and splitting simutaneously.

All plots with $tstep = 0.01$ and $\Omega = \Gamma_1, \Delta = \frac{1}{2}(\Delta_1 + \Delta_2)$

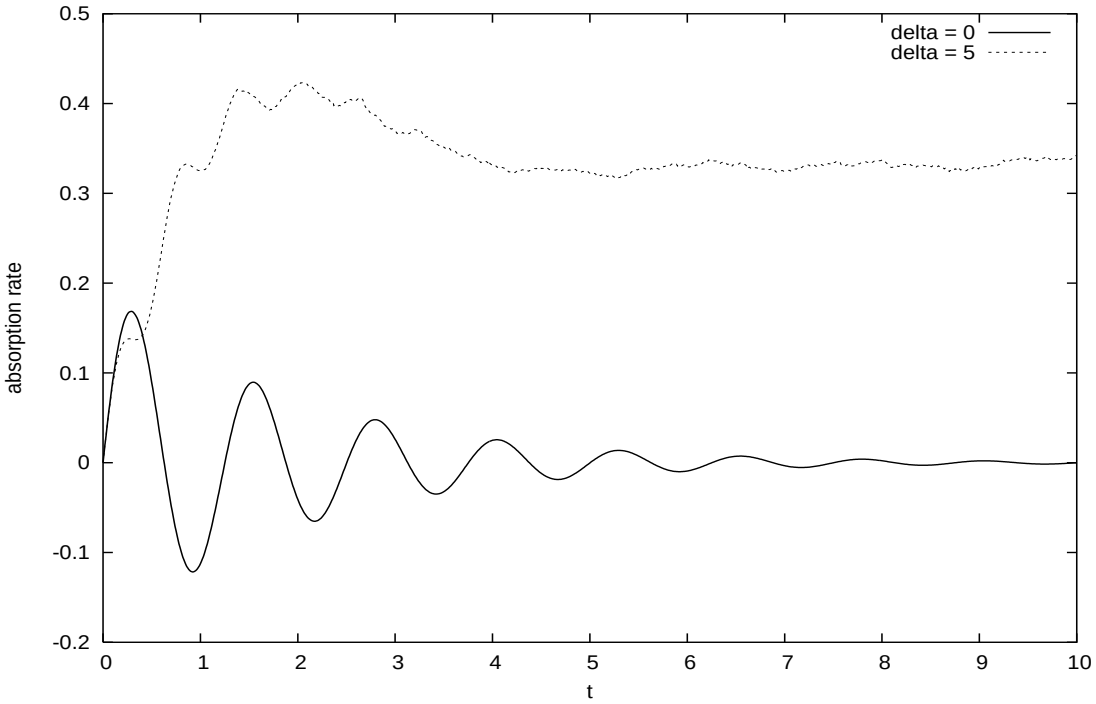

Figure 3.1: plots of absorption rate over time $\Delta = 0, 5\Gamma_1 overtime \Gamma_2 = \Gamma_1$

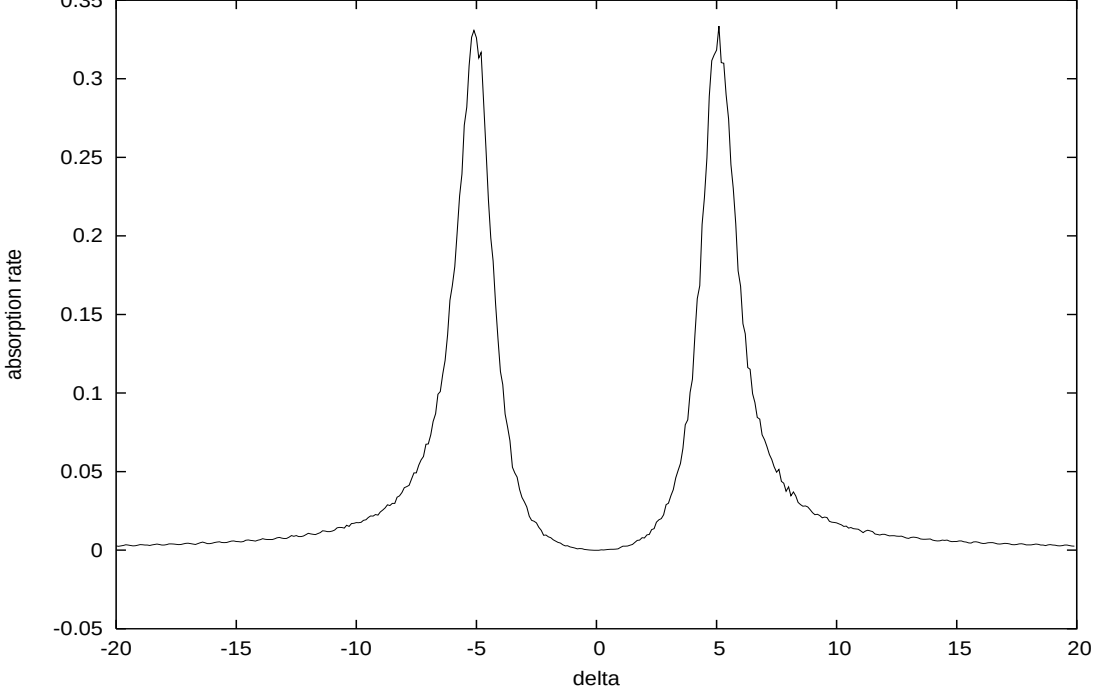

Figure 3.2: absorption rate over detuning in steady state $\Gamma_2 = \Gamma_1$

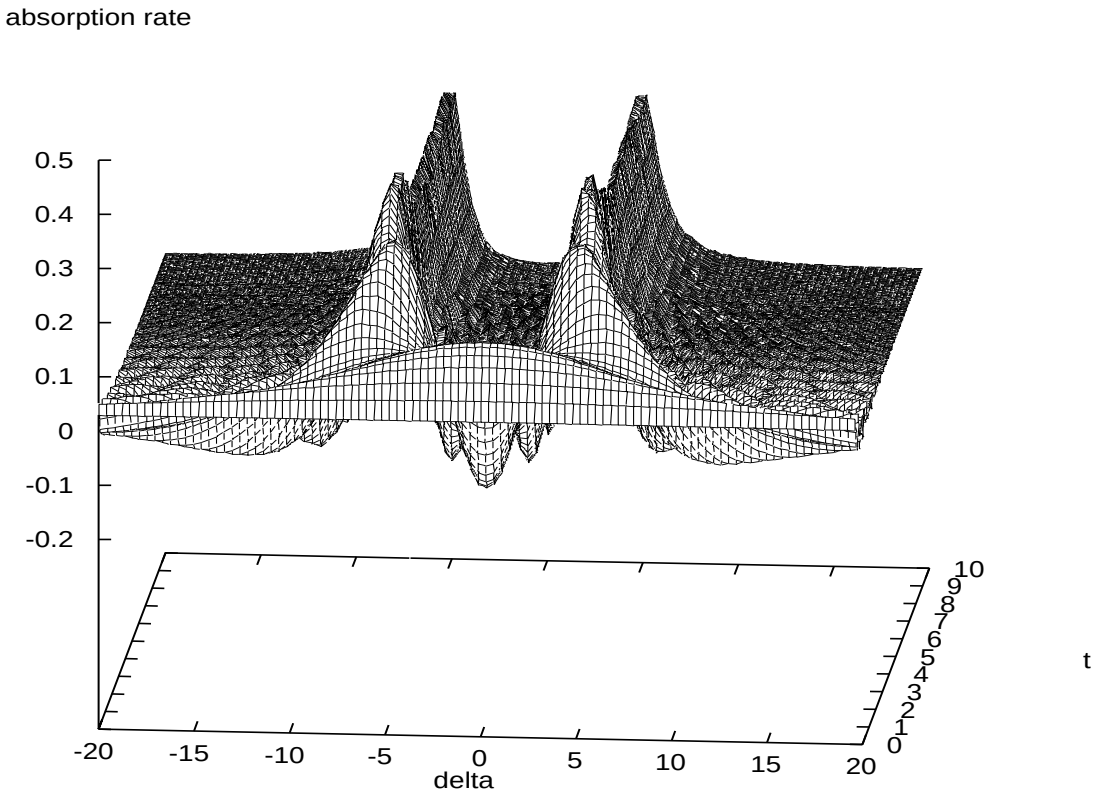

Figure 3.3: three dimensional plot of absorption rate over detuning and time looking down the 'time valley' $\Gamma_2 = \Gamma_1, \Delta = -20..+20$ in 400 steps

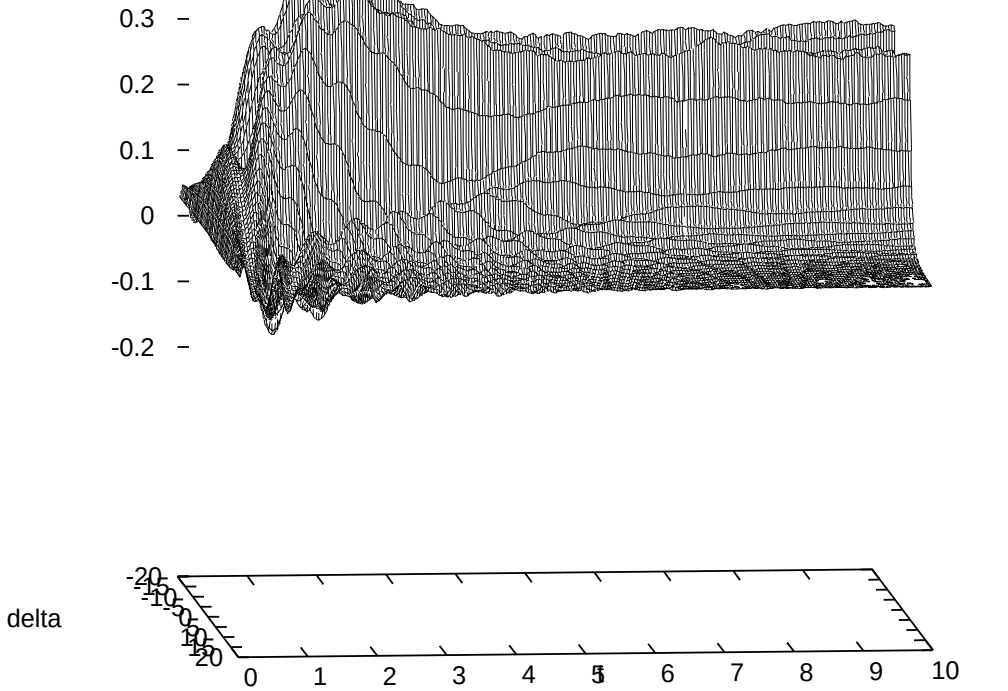

Figure 3.4: three dimensional plot of absorption rate over detuning and time looking onto from side

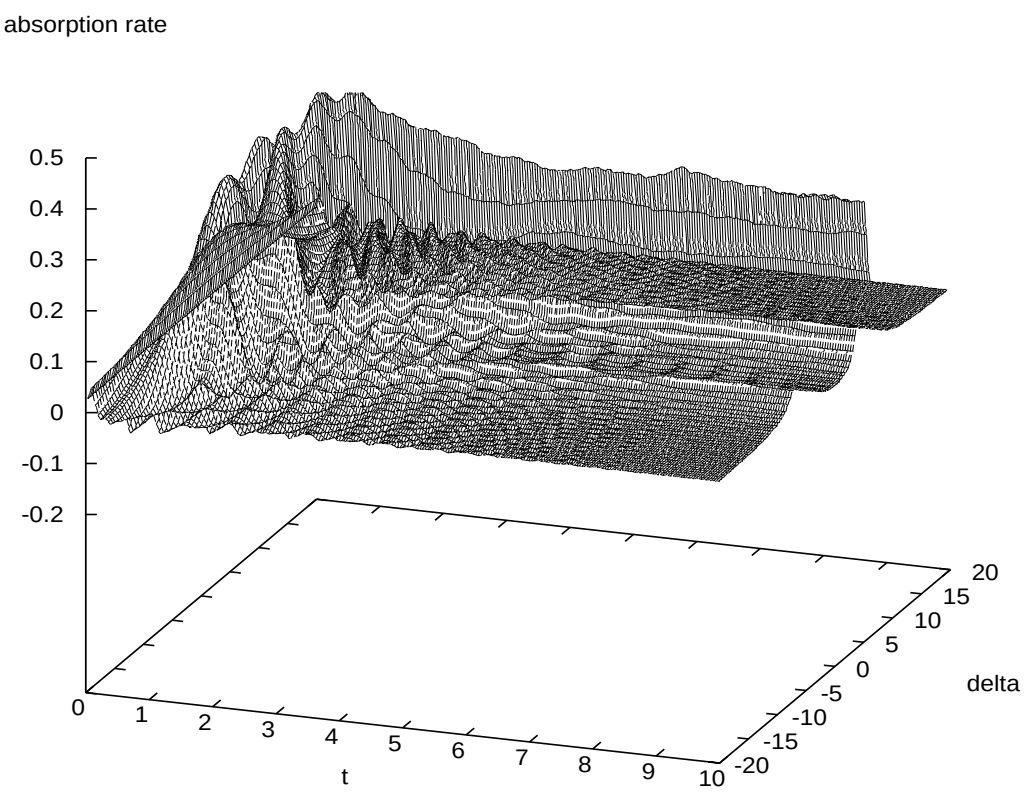

Figure 3.5: three dimensional plot of absorption rate over detuning and time looking from underneath

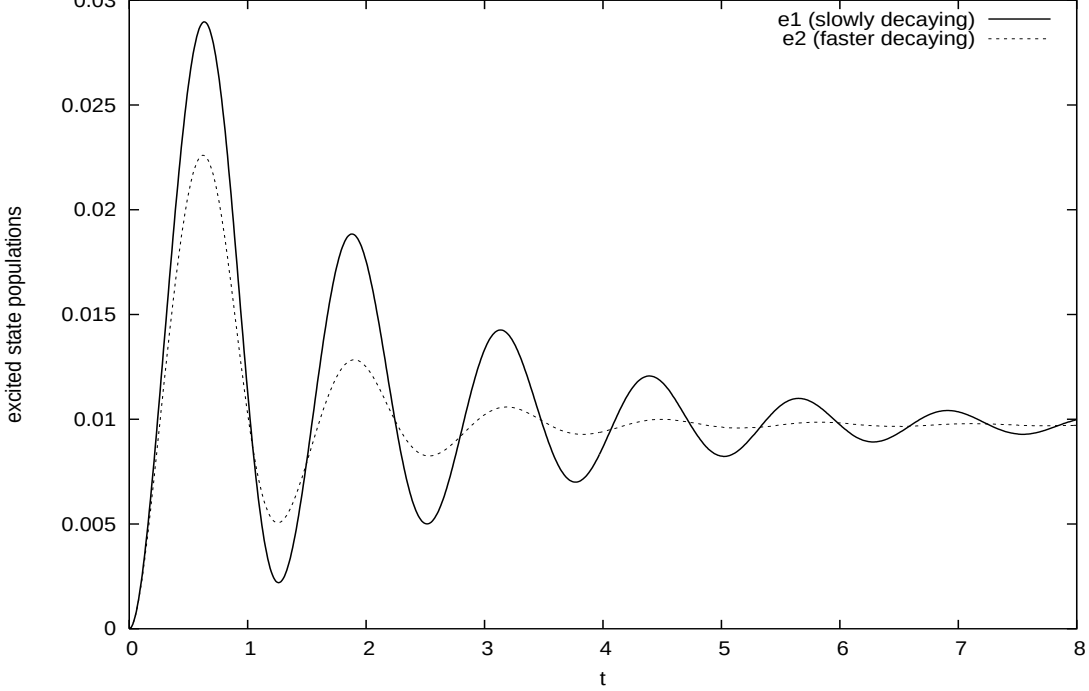

Figure 3.6: both excited state populations for different decay rates $\Gamma_2 = 2\Gamma_1, \Delta = 0$

Code: vee3 and odeab_support

0.3 vee

```
!---------------------------------------------------------------------
! !  vee.f90 ! !  Code for solving  differential equations !  using
"canned" integrator odeab.  The problem is !  the three level atom
in the vee configuration. ! !  This solution uses the stochastic
schrdinger equation !  approach which specifies equations of motion
for the !  three state vector coefficients. ! !  Output is to
standard output: first column is the detuning, !  second is time and
third the numerical solution for absorption. ! !  This code is taken
from D. Steck's example code !  for solving the harmonic oscillator
with stochastic !  reset and altered by Nima Dinyari, Xiaolu Cheng
and !  Mark Rodenberger to solve the vee atom. !
!---------------------------------------------------------------------



program vee

  ! we are using variables and subroutines in these other "modules,"
  ! so we notify the compiler to load that information
  use globals
  use odeab_support
  use odeab90
  use random_pl
  use utilities


  implicit none


  ! declare storage for solution vector; this is specific to the odeab
  ! integrator-- only modify if you want to use real variables instead
  ! of complex (change complex -> real)
  complex(wp), dimension(:), pointer :: y


  ! declare variables to use below
  real(wp)      :: t
  character(64) :: format1, buff
  integer       :: k, l, n, nout, traj


  ! declare dynamic storage array
  complex(wp), dimension(:,:), allocatable :: yout

  ! declare memory variables to calculat average decay probability
  complex(wp) :: y2old, y1old


  ! declare command-line arguments:
  !   omega  -> rabi frequency (declared in globals.f90)
  !   tstep  -> time step for integrator output
  !   tfinal -> final time of integrator output
  !   ntraj  -> number of trajectories to average together
  !   seed   -> number to seed the random number generator
  !             (same seed gives same sequence of "random" numbers)
  !             this argument is optional
  real(wp) :: tstep, tfinal
  integer  :: ntraj
  integer(rpk) :: seed  ! need to be special integer type


  !!!!!! Get arguments from the command line
  ! unfortunately this was not standardized until Fortran 2003,
  !   so these commands vary among Fortran 90/95 compilers.
  !   We will use the new standard commands, which are supported
  !   by g95; they get characters, and we use 's2r' (in utilities.f90)
  !   to convert them to real numbers.
  ! Always make sure the user's input is reasonable:
  if ( command_argument_count() .ne. 4 .&
          & and. command_argument_count() .ne. 5 ) &
     call usage()

  call get_command_argument(1, buff)
  omega = s2r(buff)

  call get_command_argument(2, buff)
  tstep = s2r(buff)
  if ( tstep .le. 0.0_wp ) then
    write(0,*)  "Error: inappropriate tstep"
    call usage()
  end if

  call get_command_argument(3, buff)
  tfinal = s2r(buff)
  if ( tfinal .le. 0.0_wp .or. tfinal .lt. tstep ) then
    write(0,*)  "Error: inappropriate tfinal"
    call usage()
  end if

  nout = nint(tfinal/tstep)
  if ( nout .gt. 5000000 ) then
    write(0,*)  "Error: too many steps!"
    call usage()
  end if

  call get_command_argument(4, buff)
  ntraj = s2i(buff)
  if ( ntraj .lt. 1 ) then
    write(0,*)  "Error: ntraj must be a positive integer"
    call usage()
  end if
  if ( ntraj .gt. 10000000 ) then
    write(0,*)  "Error: that's a crazy number of trajectories!"
    call usage()
  end if

  ! get random number seed, if specified
  if ( command_argument_count() .eq. 5 ) then
    call get_command_argument(5, buff)
    seed = s2i(buff)
    call init_rand_pl(seed1=seed)
  end if

  ! allocate output storage array (long dimension always goes first!)
  allocate( yout(nout+1, 1) )
  yout = 0


  ! the integrator needs you to do this (just leave it)
  y => odeab_y

 ! set parameters
 d = omega*10.0_wp
 delta = -20.0_wp
 gamma2 = omega * 1.0_wp



  ! main loop over detuning, goes from -20 to +20
  do n = 1, 400

  !!!!!!  Main loop over trajectories
  do traj = 1, ntraj


    ! Set initial values
    ! y(1) = ce1, y(2) = ce2, y(3) = cg
    y(1) = 0.0_wp
    y(2) = 0.0_wp
    y(3) = 1.0_wp
    y1old = y(1)
    y2old = y(2)
    t    = 0.0_wp

    ! Reset the integrator
    odeab_istate = 1

    ! Save initial condition
    yout(1, 1) = yout(1, 1) + aimag(omega*y(3)*conjg(y(1))&
                &+omega*y(3)*conjg(y(2)))

    ! Set format of output
    format1 = "(f21.15,'  ', f21.15,'  ', f21.15)"


    !!!!!! Main integration loop
    do k = 1, nout

      ! this advances the solution array 'y' from t to t+tstep
      ! note that the variable 't' gets automatically updated to t+tstep
      call odeab(t, t+tstep)

      ! if any problems were encountered, this will report it
```

```
      call handle_odeab_error()

      ! reset the trajectory with probability
      !<dagger*c>dt  (tstep must be small for
      !   good accuracy!)
     if ( rand_pl() .le. real( (0.5_wp)*((y(1)*conjg(y(1))&
                &+y1old*conjg(y1old))&
           & +gamma2*(y(2)*conjg(y(2))+y2old*conjg(y2old))&
           & +SQRT(gamma2)*(conjg(y(2))*y(1)+conjg(y2old)*y1old&
           & +conjg(y(1))*y(2)+conjg(y1old)*y2old)), wp)*tstep ) then
          y(1) = 0.0_wp
          y(2) = 0.0_wp
          y(3) = 1.0_wp
          odeab_istate = 1
          ! tell the integrator to restart the solution
      end if

      y1old = y(1)
      y2old = y(2)

      ! add results to storage array
      yout(k+1, 1) = yout(k+1, 1) + aimag(omega*y(3)*conjg(y(1))&
                     &+omega*y(3)*conjg(y(2)))

    end do  !!! main integration loop

  end do  !!! main trajectory loop


  ! divide sum by N to get average
  yout = yout / ntraj


  ! write out results
  t = 0.0_wp
  do k = 1, nout+1
    write(*,format1) delta, t, real(yout(k,1), wp)
    t = t + tstep
  end do

  delta = delta + 0.1_wp

  end do  ! main loop


  ! print performance statistics, if you're curious, to standard error
  call print_odeab_stats(cumulative = .true.)



  contains

  subroutine usage()
    ! write (0,*) means write to standard error (0), using default
    !   formatting (*)
    write(0,*) ''
    write(0,*) ''
    write(0,*) 'Usage: vee <omega> <tstep> <tfinal> <ntraj> [<seed>]'
    write(0,*) ' omega  -> scaled rabi frequency'
    write(0,*) ' tstep  -> scaled time step for integrator output'
    write(0,*) ' tfinal -> scaled final time of integrator output'
    write(0,*) ' ntraj  -> number of stochastic trajectories to average'
    write(0,*) ' seed   -> random number seed (optional)'
    write(0,*) ''
    write(0,*) 'Output is three columns of text to standard output:'
    write(0,*) '  detuning, time,  absorption rate (numerical)'
    write(0,*) ''
    write(0,*) ''
    stop
  end subroutine usage

end program vee




!---------------------------------------------------------------------
! !  Support module for hosc.f90 sample program to demonstrate the !
odeab integrator ! !  This module contains certain setup stuff for
the integrator, !    as well as the equations of motion to solve. !
!---------------------------------------------------------------------

module odeab_support

  use globals

  ! declare precision, max # of steps, and error tolerances for integration
  integer, parameter  :: odeab_prec = wp
  integer, parameter  :: odeab_maxstp = 500
  real(wp), parameter :: odeab_atol = 1.e-13_wp
  real(wp), parameter :: odeab_rtol = 1.e-13_wp

  ! the following lines declare internal storage for the odeab
  ! integrator; just leave it, except:
  !   1. change the dimension to match the number of integration variables
  !   2. if your variables are all real, change "complex"es to "real"
  type odeab_type
    complex(wp), dimension(3) :: phi
  end type
  type(odeab_type), dimension(16), save :: odeab_idx
  complex(wp), dimension(3), target :: odeab_y
  complex(wp), dimension(3), save   :: odeab_yy, odeab_p, odeab_yp
  real(wp), dimension(3), save      :: odeab_wt

  ! declare basic management stuff for integrator (just leave it)
  integer :: odeab_istate = 1
  logical, parameter :: odeab_stop = .false.


contains


! Subroutine that implements equations of motion, to be called !
by odeab; implements vee atom coefficient evolution

subroutine odeab_func(t, y, ydot)
  implicit none
  real(wp), intent(in) :: t
  complex(wp), dimension(3), intent(in)  :: y
  complex(wp), dimension(3), intent(out) :: ydot

  ! return derivative
  ! ce1dot
  ydot(1) = 0.5_wp*( i*(2*delta-d)*y(1)-i*omega*y(3)+&
            &y(1)* ( y(1)*conjg(y(1))+sqrt(gamma2)*( y(1)*conjg(y(2))+y(2)*conjg(y(1)) )&
            &+gamma2*y(2)*conjg(y(2)) )-(y(1)+sqrt(gamma2)*y(2)) )
  ! ce2dot
  ydot(2) = 0.5_wp*( i*(2*delta+d)*y(2)-i*omega*y(3)+&
            &y(2)*(y(1)*conjg(y(1))+sqrt(gamma2)*(y(1)*conjg(y(2))+y(2)*conjg(y(1)))&
            &+gamma2*y(2)*conjg(y(2)))-(gamma2*y(2)+sqrt(gamma2)*y(1)) )
  ! cgdot
  ydot(3) = 0.5_wp*( -i*omega*(y(1)+y(2))+&
            &y(3)*(y(1)*conjg(y(1))+sqrt(gamma2)*(y(1)*conjg(y(2))+y(2)*conjg(y(1)))&
            &+gamma2*y(2)*conjg(y(2))) )

  return
end subroutine odeab_func

end module odeab_support
```