Classical and Quantum Dynamics of the Driven Triangle Well

Todd Timberlake

April 30, 1997

1 Introduction

Recent experiments on the interaction between atoms and intense laser fields have led to the production of radiation at high harmonics of the laser frequency [1]. The radiation spectrum observed in these experiments displays a plateau structure. There is comparable emission at all odd harmonics up to some cut-off point at which the emission rapidly drops to zero. This behavior cannot be explained by conventional perturbation theory, which would predict that the harmonics should fall off as a simple power law of the laser intensity. Theoretical studies of simple laser-atom systems have qualitatively reproduced this high harmonic generation (HHG), but further study of simple quantum wells driven by a strong sinusoidal field is needed.

One very simple quantum system that exhibits this type of behavior is the driven triangle well. This sytem differs from a true atomic system is a few important aspects. First, the triangle well is not symmetric under a parity transformation. It is precisely this symmetry that forbids the production of even order harmonics in atomic systems. Thus, the driven triangle well will radiate at both even and odd harmonics. Also, the triangle well is a completely bound system allowing no ionization. Multi-photon ionization processes have been a critical part of the theoretical explanation of HHG in atomic systems. However, the driven triangle well illustrates that a system which cannot ionize can still show HHG. Aside from these differences the triangle well should prove to be a reasonable (if somewhat simplistic) model for laser-atom interactions in which bound states, and not continuum states, play the most important role.

We calculate radiation spectra for the driven triangle well using various values of the driving field strength. The onset of HHG is observed. The dynamics of the corresponding classical system is also studied, using parameters indentical to those used in computing the quantum radiation spectra. There appears to be a strong connection between the onset of chaos in the classical system and HHG in the quantum system, as shown by Cocke and Reichl [2]. This connection may apply to more realistic atomic models and may lead to a deeper understanding of the processes that cause HHG.

To further study this connection between HHG and classical chaos one immediately turns to semiclassical methods. Semiclassical methods have proven very useful in calculating radiation spectra for atomic models that allow ionization [3], but at this time there is no corresponding semiclassical method for totally bound systems. One other area in which semiclassical methods have enjoyed recent sucess is in the calculation of the energy density for states of quantum systems [4]. Of course, the driven triangle well is an open system and thus has no energy eigenvalues. However, *quasienergies* play a role in time-periodic systems that is similar to that of energies in conservative systems. Tabor [5] has developed a semiclassical method for computing the quasienergies of area-preserving maps. This method can be extended to provide a semiclassical theory of quasienergies for driven one-dimensional systems. Preliminary work on the driven harmonic oscillator indicates that this method has some promise in linking the classical dynamics (in terms of classical periodic orbits) and quantum dynamics (in terms of quasienergies) for these types of systems. Application of this theory to the driven triangle well is currently under way.

2 Classical Motion

Our model consists of a linear potential that terminates in an inifinite wall. This system is then driven by a classical radiation field with dipole coupling to the particle. The Hamiltonian for the system is

$$\begin{aligned} H &= \frac{p^2}{2m} - \epsilon_0 x - \epsilon x \cos(\omega t + \theta) + V_L(x) \qquad (1) \\ V_L(x) &= \begin{cases} 0 & \text{if } x < L \\ \infty & \text{if } x > L \end{cases} \end{aligned}$$

Here ϵ_0 is the strength of the linear potential, ϵ is the strength of the driving field, ω is the frequency of the driving field, and θ is the phase of the driving field at $t = 0$. The system is integrable without the driving field or without the infinite wall. However, if both are present the driving term will give rise to an infinite series of non-linear resonances in the phase space. As ϵ is increased these resonances get larger and may begin to overlap. Overlap of the resonances destroys the invariant tori in the phase space and the regions of overlap will become chaotic. At a high enough value of ϵ the phase space will be globally chaotic.

One good way to view this transition to chaos is to plot a collision map for the system. The collision map simply maps coordinates of the system at a collision with the infinite wall to the coordinates at the next collision. By iterating this map for several initial conditions one can get an accurate description of the dynamics for this system [6]. One pair of canonically conjugate coordinates that can be used for this map is E_0, the energy of the undriven triangle well, and ϕ, the phase of the driving field. For convenience we will use J, the canonical action, and ϕ. This still gives us an accurate description of the dynamics because

$$J = \frac{2\sqrt{2m}}{3\pi\epsilon_0}(E_0 + \epsilon_0 L)^{3/2}. \qquad (2)$$

To create such a plot one must simply integrate the equations of motion between collisions with the wall. These take the very simple form

$$\begin{aligned} \dot{x} &= \frac{p}{m} \qquad (3) \\ \dot{p} &= \epsilon_0 + \epsilon \cos(\omega t + \theta) \end{aligned}$$

which can easily be numerically integrated with any ODE solver. However, we want to reverse the momentum when x reaches L. Thus we need an ODE integrator that has a root-finding feature. The program 'twcmap.f' (see Source Code) uses routine from the NAG software library, 'do2cje', which has such a feature. The program simply integrates the ODEs until $x = L$, plots the values of J and ϕ, reverses the momentum, and continues the integration until the next collision. J can be written in terms of x and p by using (2) and

$$E_0 = \frac{p^2}{2m} - \epsilon_0 x. \qquad (4)$$

At the wall $x = L$, so the formula for J simplifies to

$$J = \frac{p^3}{3\pi\epsilon_0 m}. \qquad (5)$$

One can see from Figure 1 that the system becomes chaotic as ϵ is increased from 0.01 to 0.3. For $\epsilon = 0.01$ the plot consists solely of invariant tori. For $\epsilon = 0.1$ the resonances are begining to overlap, leading to the production of 'daughter resonances' and the breakup of the tori in the regions of overlap. At $\epsilon = 0.2$ and 0.3 there are still islands of stability, but the plot is dominated by chaos.

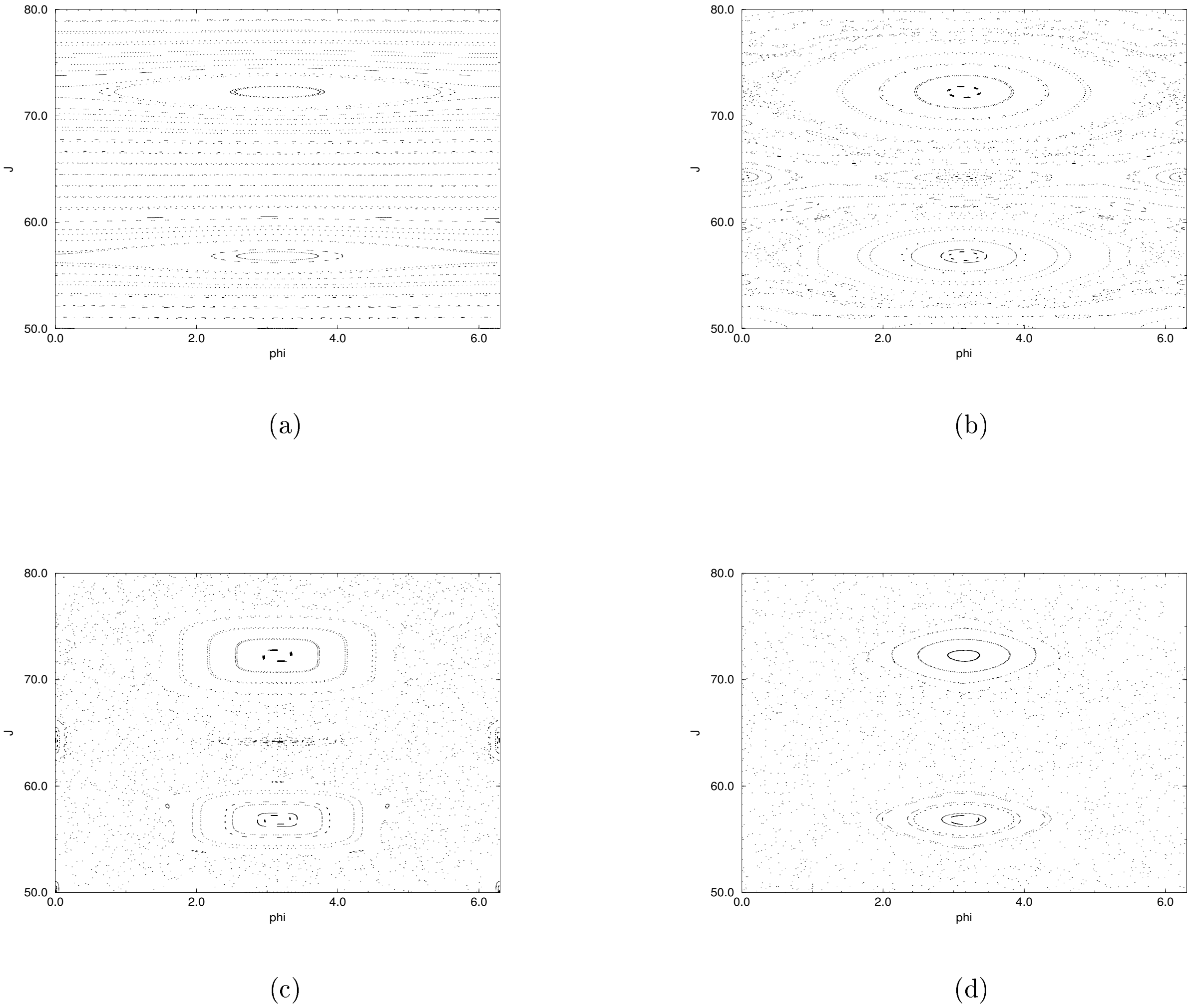


Figure 1: Collision maps of the classical system for (a) $\epsilon = 0.01$, (b) $\epsilon = 0.1$, (c) $\epsilon = 0.2$, and (d) $\epsilon = 0.3$. For all four plots $\omega = 2.52$, $\epsilon_0 = 0.4$, and $L = 600$.

3 Quantum Dynamics

Now we consider the quantum dynamics of the driven triangle well. First we will look at the triangle well without the driving field. The undriven triangle well is an analytically solveable system. The time-independent Schrödinger's equation for that system is

$$-\frac{1}{2m}\frac{\partial^2 \psi_i}{\partial x^2} = (E_i + \epsilon_0 x)\,\psi_i \quad \text{for } x \leq L. \tag{6}$$

The solution is an Airy function:

$$\psi_i(x) = \langle x|\psi_i\rangle = N_i \mathrm{Ai}\left[-(\tfrac{2m}{\hbar^2})^{1/3}\epsilon_0^{-2/3}E_i - (\tfrac{2m\epsilon_0}{\hbar^2})^{1/3}x\right] \tag{7}$$

The energy eigenvalues for the undriven system are obtained by making the wavefunctions $\psi_i(x)$ go to zero at $x = L$. Thus, the eigenvalues are related to the zeros of the Airy function. The nth zero of $\mathrm{Ai}(x) \propto n^{2/3}$ for large n. Thus, $E_i \propto i^{2/3}$, which agrees with the classical result $E_0 \propto J^{2/3}$. The program 'triwev.f' (see Source Code) uses Newton's method to solve for the zeros of the Airy function and then uses the results to calculate the energy eigenvalues for the undriven triangle well. This program was used to compute the first thousand eigenvalues using the parameters $\epsilon_0 = 0.4$ and $L = 600$.

Once we turn on the driving field the system becomes non-integrable. Therefore we must numerically calculate its behavior. This could be done with a finite-difference technique that solves the time-dependent Schrödinger's equation for the system,

$$\begin{aligned} i\hbar\frac{\partial}{\partial t}|\psi(t)\rangle &= [H_0 - \xi(t)x\cos(\omega t+\theta)]\,|\psi(t)\rangle \\ H_0 &= \frac{p^2}{2m} - \epsilon_0 x + V_L(x) \\ \xi(t) &= \begin{cases} \epsilon\sin^2(\frac{\omega t}{4n}) & t < \frac{2\pi n}{\omega} \\ \epsilon & t > \frac{2\pi n}{\omega} \end{cases} \end{aligned} \tag{8}$$

where $\xi(t)$ is a turn-on function that simulates the finite rise-time of the laser field amplitude. However, an alternative method offers more insight and a very simple way to calculate radiation spectra. This alternative is to decompose the wavefunction into eigenstates of the undriven system. We write this decomposition as

$$\psi(t) = \sum_i c_i(t)|\psi_i\rangle \tag{9}$$

where the c_i's are complex, time-dependent coefficients. Writing the wavefunction in this way allows us to directly study the diffusion of energy in this system. It also reduces the numerical calculation to an initial value ODE problem, albeit with a large number of ODE's.

The ODE's for the coefficients are given by

$$\begin{aligned} \frac{\partial c_i(t)}{\partial t} &= -\frac{iE_i}{\hbar}c_i(t) + \frac{i}{\hbar}\xi(t)\cos(\omega t+\theta)\sum_j x_{ij}c_j(t) \\ x_{ij} &= \begin{cases} -\frac{2E_i}{3\epsilon_0} & i = j \\ -\frac{\epsilon_0}{m(E_i-E_j)^2} & i \neq j \end{cases} \end{aligned} \tag{10}$$

where the x_{ij}'s have been taken from [6].

To calculate radiation spectra we must first calculate a time series for the expectation value of the acceleration. It has been shown that using the expectation value of acceleration is superior to using the dipole moment for problems in which the final position and velocity of the particle are non-zero [2]. The equation for the expectation value of $\ddot{x}$ takes a particularly simple form when we decompose the wavefunction into energy eigenstates of the undriven system. The expectation value is given by

$$\begin{aligned} \langle\psi(t)|\ddot{x}|\psi(t)\rangle &= \sum_{ij} c_i^*(t)\ddot{x}_{ij}c_j(t) \\ \ddot{x}_{ij} &= -\frac{1}{\hbar^2}\left(E_i - E_j\right)^2 x_{ij} + \frac{1}{\hbar^2}\xi(t)\cos(\omega t+\theta)\sum_k \left(2E_k - E_j - E_i\right) x_{ik}x_{kj} \end{aligned} \tag{11}$$

The program 'accel.f' (see Source Code) integrates the ODEs for the $c_i(t)$'s using an initial condition of $c_{inst}(0) = 1$ and $c_i(0) = 0$ for $i \neq inst$. $inst$ is an initial state that is specified in the source code. In this paper we always use $inst = 60$. The total integration time (in cycles of the driving field) is given as a command line argument, as are the values of ω and ϵ and the maximum harmonic order to be viewed in the spectrum. At each output time 'accel.f' outputs the expectation value of $\ddot{x}$. The program can optionally output the variance of the energy distribution (Δn^2) and the probability for each n ($|c_n(t)|^2$). The ODEs are integrated using the routine 'lsoda' which has a convenient way to pass energy eigenvalues and dipole matrix elements to the derivative-evaluating subroutine.

Once the acceleration time series has been calculated all that remains is to take a Fourier transform. The acceleration time series $\ddot{x}(t)$ is related to the power spectrum by the equation

$$X(\omega) = \frac{1}{\sqrt{2\pi}}\int_0^T dt\, e^{-i\omega t}\ddot{x}(t) \tag{12}$$

where T is the total time for which $\ddot{x}(t)$ has been computed. The program 'powspec.f' (see Source Code) uses a Fast-Fourier Transform routine from Numerical Recipes in FORTRAN 77 [7], to take the Fourier transform of the output from 'accel.f'. Radiation spectra for the driven triangle well system are shown in Figure 2. The onset of HHG as ϵ is increased is evident from these spectra. Moreover, the onset of HHG appears to occur at a value of ϵ similar to that at which the classical motion becomes chaotic (see Figure 1).

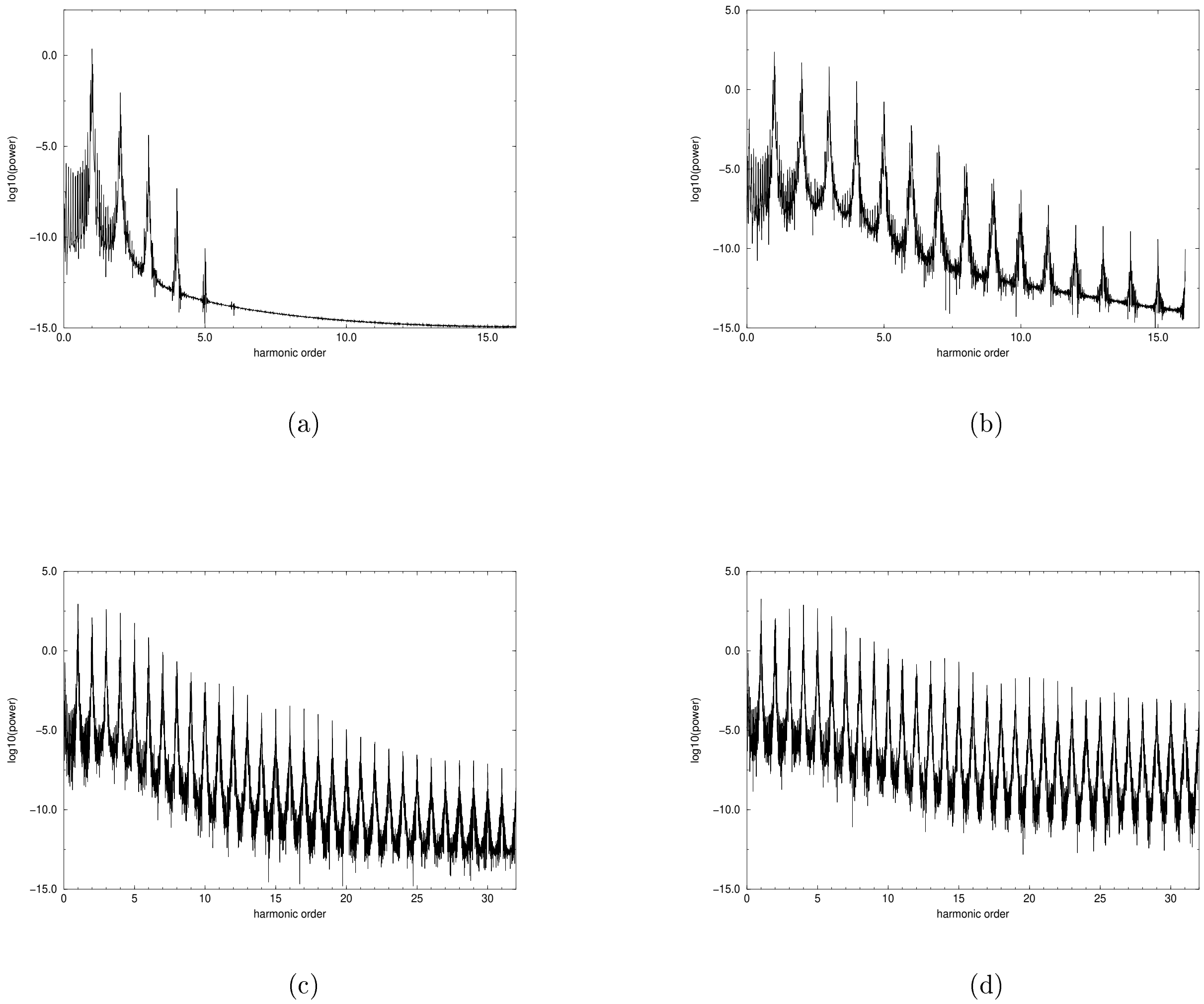


Figure 2: Power spectra for the driven triangle well with (a) $\epsilon = 0.01$, (b) $\epsilon = 0.1$, (c) $\epsilon = 0.2$, and (d) $\epsilon = 0.3$. All spectra were calculated using $\omega = 2.52$, $\epsilon_0 = 0.4$, $inst = 60$, a twelve cycle turn-on ($n = 12$ in the definition of $\xi(t)$), and a basis of 400 eigenstates.

4 Quasienergies

This apparent connection between chaos in the classical motion and high harmonic generation in the radiation spectra of the driven triangle well provides ample encouragement for further study of the connection between classical and quantum dynamics in this system. The most direct way to study this connection is to use semiclassical methods to explore the behavior of the quantum system. These semiclassical methods use only information about the classical motion of the system, yet in many cases they give an accurate description of the quantum dynamics.

The best way to study the connection between chaos and HHG would be to use semiclassical methods to directly calculate radiation spectra. This has been done on driven, one-dimensional systems that allow ionization [3]. This techniques uses a quantum calculation of an ionization probability, followed by semiclassical propagation of the free electron in the radiation field, and finally another quantum calculation of the recombination probability. Thus, the semiclassical method is only applied while the electron is ionized. This technique does produce spectra with HHG in some atomic models. However, since the driven triangle well cannot ionize, this method cannot be directly applied to our problem. At this time, there is no known semiclassical technique for calculating radiation spectra for this type of system.

Since we cannot calculate the radiation spectra, we must find another type of semiclassical analysis that may help us understand the connection between HHG and chaos. In conservative systems semiclassical methods have been used to calculate densities of energy states with great success [4]. The method involves summing a series of terms, each of which corresponds to a periodic orbit of the classical motion, to obtain a formula for the energy density of states. One can then study the level spacing statistics and other properties of the density of states to gain insight into how the classical periodic orbits affect the quantum problem.

Of course, the driven triangle well is an open system with no energy eigenvalues. Thus, it might at first seem that periodic orbit methods are useless for such a system. However, time-periodic systems do have quasienergies. Quasienergies play a role in time-periodic systems that is very similar to the role of energies in conservative systems. In fact, the energy density of states is just a Fourier transform of a quasienergy density of states in the limit as the period of the system goes to zero. We expect the quasienergy density of states to be as fruitful an analytic tool for time-periodic systems as the energy density of states is for conservative systems.

Quasienergies are related to the eigenvalues of the Floquet operator, which propagates the system forward in time by one period T. Thus,

$$\hat{U}_T|\psi(t)\rangle \quad = \quad |\psi(t+T)\rangle. \tag{13}$$

If the system is periodic with period T then $\hat{U}_T$ can be diagonalized. The eigenvalues of the Floquet operator are all of unit modulus (the Floquet operator is unitary by definition), and can be written in the form $e^{-\frac{iq_nT}{\hbar}}$. The q_n are the quasienergies of the system. The are only defined modulo $\omega\hbar$ (if q_n is a quasienergy, then so is $q_n + \omega\hbar$).

5 Periodic Orbits and Quasienergies

5.1 Periodic Orbit Theory of Quasienergies

To construct the quasienergy density of states in terms of periodic orbits, we first note the identity

$$\begin{aligned} \mathrm{Tr}\left(\hat{U}_{NT}\right) &= \sum_n \langle n|\hat{U}_{NT}|n\rangle \\ &= \sum_n e^{-\frac{iNq_nT}{\hbar}} \end{aligned} \tag{14}$$

where $|n\rangle$ is a quasienergy eigenstate (i.e. $\hat{U}_{NT}|n\rangle = e^{-iNq_nT}|n\rangle$). We then construct the quantity

$$\begin{aligned}\rho(Q) &= \frac{T}{2\pi\hbar}\sum_{N=-\infty}^{\infty} e^{\frac{iNQT}{\hbar}}\,\mathrm{Tr}\left(\hat{U}_{NT}\right) \qquad (15)\\ &= \frac{T}{2\pi\hbar}\sum_{N=-\infty}^{\infty}\sum_{n} e^{\frac{iNT(Q-q_n)}{\hbar}}\end{aligned}$$

and then use the identity

$$\sum_{N=-\infty}^{\infty} e^{2\pi iNx} = \sum_{M=-\infty}^{\infty}\delta(x-M).$$

With a little manipulation we find

$$\rho(Q) = \sum_{M=-\infty}^{\infty}\sum_{n}\delta(Q-(q_n+M\omega\hbar)) \qquad (16)$$

where $\omega = \frac{2\pi}{T}$. Thus we see that the quantity $\rho(Q)$ is exactly the quasienergy density of states (a delta function at each quasienergy mod ω).

The connection to periodic orbits comes about because we can think of $\hat{U}_{NT}$ as being a path integral from $t = 0$ to $t = NT$. The operator itself can be thought of as a sum over all possible paths of duration NT. However, the trace of the operator will involve only periodic orbits whose period is NT. Thus we see that the quasienergy density of states can be written as a sum of terms, each of which corresponds to a periodic orbit whose period is a multiple of the driving period. Tabor [5] has developed a very simple method for calculating the quasienergy density of states for an area-preserving map using the periodic orbits of the map. This technique can be extended to time-periodic, one-dimensional systems by consider the map to be the integration of the equations of motion over one period. Tabors method consists of the following steps:

1. Locate all periodic orbits of period NT.
2. Calculate the stability matrix $\mathbf{M_j}$ for each periodic orbit.

$$\mathbf{M_j} = \prod_{i=0}^{N}\begin{bmatrix}\frac{dp_{i+1}}{dp_i} & \frac{dp_{i+1}}{dx_i}\\ \frac{dx_{i+1}}{dp_i} & \frac{dx_{i+1}}{dx_i}\end{bmatrix}_j \qquad (17)$$

3. Calculate the residue R_j for each periodic orbit.

$$R_j = \frac{1}{4}(2-\mathrm{Tr}\mathbf{M_j}) \qquad (18)$$

4. Calculate the classical action $\bar{W}_j$ along each orbit.

$$\begin{aligned}\bar{W}_j &= \int_0^{NT} L_j(t)\,dt \qquad (19)\\ &= \int_0^{NT}(p_j(t)\dot{x}_j(t)-H_j(t))\,dt\end{aligned}$$

5. Repeat above procedure for all N.

6. Calculate the quasienergy density of states.

$$\rho(Q) \; = \; \frac{T}{2\pi\hbar}\sum_j \frac{1}{2\sqrt{R_j}} e^{(i/\hbar)(N_j QT+\bar{W}_j)} \tag{20}$$

Of course, one cannot actually perform the above procedure for all N since N ranges from $-\infty$ to ∞. Thus, one must truncate the series at some maximum value of $|N|$. If the series does not naturally converge, then a damping term can be added that will lead to a convergent series and a finite resolution density of states [8].

5.2 Results for the Driven Harmonic Oscillator

We can apply this technique to the driven harmonic oscillator. The advantage of this system is that it is integrable and we can analytically find the periodic orbits, calculate their actions and residues, and construct the quasienergy density of states. The program 'orbitnt.f' (see Source Code) numerically finds the periodic orbits of the driven harmonics oscillator, computes their residues and actions, and optionally generates a plot of the periodic orbits. This program uses the 'lsoda' ODE integrator to integrate the equations of motion form $t = 0$ to $t = NT$ and then uses Newton's method to adjust the initial conditions until a periodic orbit is found. The map of the coordinates over one cycle of the driving field is linear, so only one Newton iteration is required (although in practice there will be two or three since the derivatives that the program calculates are only approximate). Also, the driven harmonic oscillator has only one periodic orbit for each N if the oscillator frequency is incommensurate with the driving frequency. A plot of this orbit is shown in Figure 3. 'orbitnt.f' was used to find the residues and actions of the first 60 repetitions of that periodic orbit using $\omega_0 = 4$ and $\omega = 4\pi$, where ω_0 is the oscillator frequency. The programs 'dhonqp.f' and 'dhoqp.f' were used to produce plots of the numerical and analytical quasienergy density of states, respectively. These plots, and the error in the numerical calculation, are shown in Figure 4.

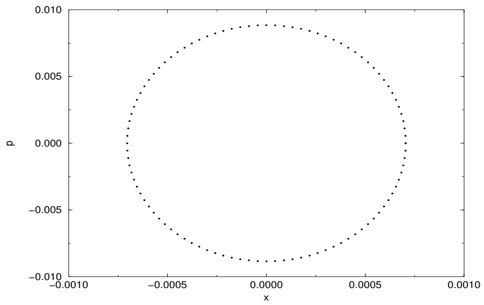
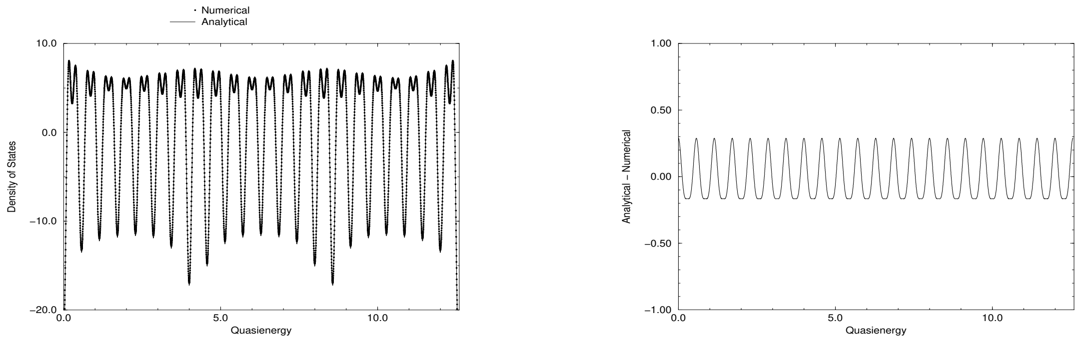


Figure 3: The only periodic orbit of the driven harmonic oscillator whose period is an integer multiple of the driving period. This orbit was calculated using $\omega_0 = 4$ and $\omega = 4\pi$.



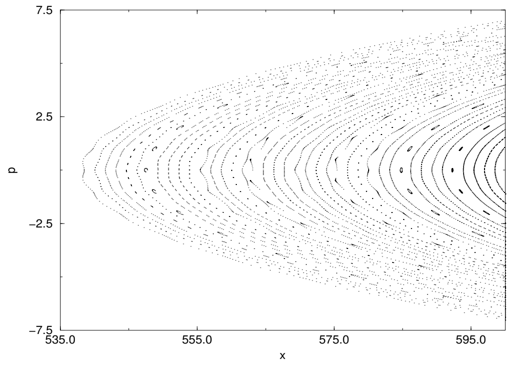
(a) (b)

Figure 4: Comparison of numerical and analytical quasienergy density of states for the driven harmonic oscillator. The plots show (a) the numerical and analytical results, and (b) the difference between the two.

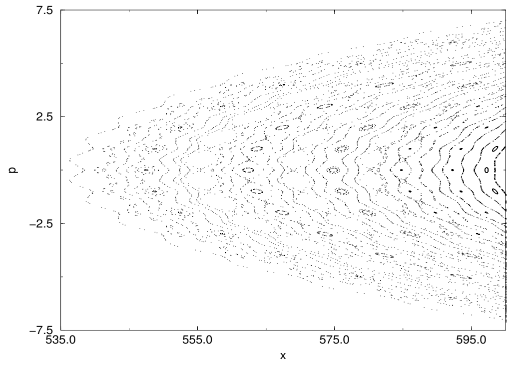
5.3 Application to the Driven Triangle Well

Unfortunately, applying this method to the driven triangle well is not so easy as for the driven harmonic oscillator. There are several complications that arise in the driven triangle well system. The first is the problem of the infinite wall. Again we need an ODE integrator that has a root-finding feature. However, this time we also need the integrator to continue integrating after it finds a root (e.g. hits the infinite wall) until some specified time. At this point we have not found a pre-packaged integrator that will perform this function. If none can be found it should be possible to write a new ODE routine that has this feature. The next problem with the triangle well is that the map $t \to t+T$ is not linear in the initial coordinates as it was for the harmonic oscillator. This means that several Newton iterations may be required to locate a single periodic orbit. Another difficulty is that there may be several periodic orbits at a given period NT. The basin of attraction for each orbit will be finite, so our initial point must start out relatively close to a periodic orbit.

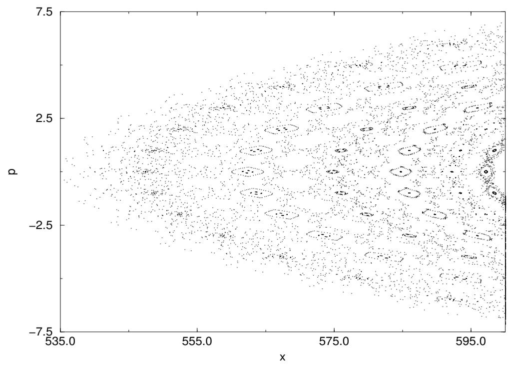
This problem of multiple orbits with finite bases of attraction can be seen in strobe plots of the phase space for this system. The strobe plots are just plots of the iteration of the map $t \to t+T$. The program 'twsplt.f' (see Source Code) generates strobe plots for the driven triangle well system. Strobe plots are shown in Figure 5 for various values of ϵ. Each string of resonances corresponds to a different periodic orbit, whose basin of attraction is related to the size of the resonance. This difficulty can be solved by using a mesh of initial points in the phase space. For a fine enough mesh there should be at least one initial point that is in the basin of attraction of each periodic orbit (and thus close enough that only a few Newton iterations will be required to reach the periodic orbit). An attempt to use this method to calculate the quasienergy density of states for the driven triangle well is currently underway.



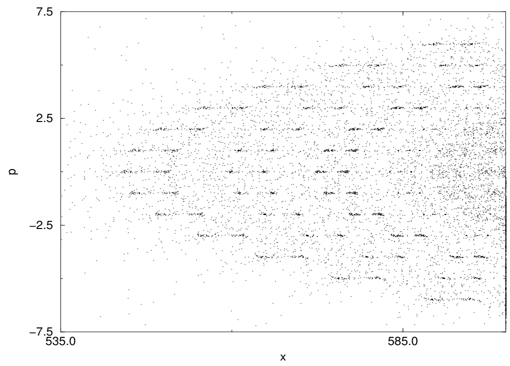
(a)



(b)



(c)



(d)

Figure 5: Strobe plots of the phase space for the driven triangle well with (a) $\epsilon = 0.01$, (b) $\epsilon = 0.1$, (c) $\epsilon = 0.2$, and (d) $\epsilon = 0.5$. For all four plots $\omega = 2.52$, $\epsilon_0 = 0.4$, and $L = 600$.

6 Conclusion

As we have seen, there appears to be a strong connection between chaos in the classical motion of the driven triangle well and HHG in the radiation spectrum of the quantum system. At this point the connection is a phenomenological one only. There is no clear reason for why there should be a connection between chaos and HHG in this type of system. Direct semiclassical calculation of the radiation spectrum might reveal the reason for this connection, but at present there is no satisfactory semiclassical method for calculating radiation spectra for totally bound systems. Thus, one must resort to alternative methods for connecting the classical and quantum dynamics. Some success has been achieved in developing a semiclassical technique for calculating the quasienergy density of states for the driven harmonic oscillator. It is hoped that this method can be extended to the driven triangle well. This would provide a direct link between properties of the classical motion (periodic orbits) and properties of the quantum dynamics (quasienergies). Further analysis may give some insight into how the classical motion influences the quantum radiation spectrum of this system. Furthermore, if this method proves successful for the driven triangle well, then it is likely that it can be used to study other one-dimensional systems in which the bound states play a dominant role in the dynamics. In this way, we may come to understand the connection between chaos and HHG in realistic models of atoms.

7 Source Code

7.1 Collision Maps

```
c=======================================================================
c     history: twsplt.f
c     twcmap:  This program generates a map between successive
c     collisions with the infinite wall in the driven
c     triangle well system.  This system has a Hamiltonian
c     H = p**2/(2*m) - V0*x - V*x*cos(w*t+theta)     x < 600
c     Because of the presence of the infinite wall the program
c     do2cje from the NAG library is used.  This program integrates
c     the equations of motion until the particle hits the wall.
c     Values of J (action) and phi (phase) are written to standard out.
c     The momentum is then reversed and the equations of motion
c     are again integrated until another collision, etc.
c=======================================================================
      program twcmap

      implicit none

c-----------------------------------------------------------------------
c     n: number of odes
c     iwork: size of work array used by d02ejf
c-----------------------------------------------------------------------
      integer n, iwork
      parameter (n = 2, iwork = 28 + 21*n)

c-----------------------------------------------------------------------
c     tbgn: initial time of integration (overwritten by do2cje)
c     tend: final time
c     tol: tolerance for d02cje
c-----------------------------------------------------------------------
      real   tbgn, tend, tdt, dt
      parameter (dt = 1.0e-4)
      real   tol, default_tol
      parameter (default_tol = 1.0e-8)

c-----------------------------------------------------------------------
c     i: index used to increment output times
c     ifail: error return from do2cje
c-----------------------------------------------------------------------
      integer ifail

c-----------------------------------------------------------------------
c     work: work array used by d02cje
c     y: solution vector
c     g: function that monitors collisions with the wall
c     J: canonical action variable
c     phi: phase of driving field
c-----------------------------------------------------------------------
      real   work(iwork), y(n)
      real   g, j, phi

c-----------------------------------------------------------------------
c     Jmax: maximum value of J used as initial condition
c     Jmin: minimum value of J "
c     nphi: number of initial values of phi
c     nJ: "                              J
c     niter: number of strobe times (approx)
c-----------------------------------------------------------------------
      real jmax, jmin
      integer nphi, nj, niter
      integer iphi, ij, iter

c-----------------------------------------------------------------------
c     argument parsing variables
c-----------------------------------------------------------------------
      real r8arg, r8_never
      parameter (r8_never = -1.0e-60)
      integer iargc

c-----------------------------------------------------------------------
c     fcn: subroutine that calculates deriviatives
c     pederv: "    "     "  Jacobian
c     out: subroutine that outputs solution at specified times
c-----------------------------------------------------------------------
      external g, d02cje, fcn, cjwd02, cjxd02

c-----------------------------------------------------------------------
c     Common communication with subroutines
c-----------------------------------------------------------------------
      include 'fcn.inc'

      real   pi

c-----------------------------------------------------------------------
c     parse command line arguments
c-----------------------------------------------------------------------
      if (iargc() .lt. 2) go to 900

      w = r8arg(1, r8_never)
      v = r8arg(2, r8_never)
      if (w .eq. r8_never .or. v .eq. r8_never) go to 900

      tol = r8arg(3, default_tol)
      if (tol .le. 0.0e0) tol = default_tol

c-----------------------------------------------------------------------
c     Set values of parameters
c-----------------------------------------------------------------------
      v0 = 4.0e-1
      m = 1.0e0
      niter = 100
      pi = 4.0e0*atan(1.0e0)
      per = 2.0e0*pi/w
      tend = 1.0e3*niter*per
      l = 6.0e2

c-----------------------------------------------------------------------
c     set iteration parameters
c-----------------------------------------------------------------------
      jmax = 80
      jmin = 50
      nphi = 2
      nj = 30

c-----------------------------------------------------------------------
c     iterate over initial conditions
c-----------------------------------------------------------------------
      do iphi = 1, nphi

         do ij = 1, nj

c-----------------------------------------------------------------------
c     initialize values of theta and J
c-----------------------------------------------------------------------
            theta = 0.0e0+real(iphi-1)*pi/real(nphi-1)
            j = jmin + real(ij-1)*(jmax-jmin)/real(nj-1)

c-----------------------------------------------------------------------
c     Initialize solution at t = 0 (y(1) = x, y(2) = p)
c-----------------------------------------------------------------------
            tbgn = 0.0e0
            y(1) = l
            y(2) = (3.0e0*pi*v0*m*j)**(1.0e0/3.0e0)

c-----------------------------------------------------------------------
c     begin loop over collisions with wall
c-----------------------------------------------------------------------
            do iter = 1, niter

c-----------------------------------------------------------------------
c     Call d02ejf to integrate y until collsion with wall
c-----------------------------------------------------------------------
               ifail = 0

               call d02cje(tbgn,tend,n,y,fcn,tol,'D',cjxd02,g,
     &              work,ifail)

c-----------------------------------------------------------------------
c     Error message if d02cje fails
c-----------------------------------------------------------------------
               if (ifail .gt. 0) then
                  write(0,*) 'd02cje returns ifail = ',ifail
                  stop
               end if

c-----------------------------------------------------------------------
c     Reverse momentum and return to d02cje unless at the end
c-----------------------------------------------------------------------
               j = y(2)*y(2)*y(2)/(3.0e0*pi*v0*m)
               phi = mod(w*tbgn + theta, 2.0e0*pi)
               write(*,*) phi, j
               y(2) = -y(2)
               ifail = 0
               tdt = tbgn + dt
               call d02cje(tbgn,tdt,n,y,fcn,tol,'D',cjxd02,cjwd02,
     &              work,ifail)
            end do
         end do
      end do

      stop

c-----------------------------------------------------------------------
c     usage message
c-----------------------------------------------------------------------
 900  continue
```

```
      write(0,*) 'usage: twsplt <w> <V> [<tol>]'
      write(0,*)
      write(0,*) 'w: frequency of driving field'
      write(0,*) 'V: strength of driving field'
      write(0,*) 'tol: optional tolerance for d02cje'

      stop

      end

c=======================================================================
c     This routine defines the equations of motion for the driven
c     triangle well.  It is used with twcmap.f
c=======================================================================
      subroutine fcn(t, y, f)

      implicit none

c-----------------------------------------------------------------------
c     common communication with main program
c-----------------------------------------------------------------------
      include 'fcn.inc'

c-----------------------------------------------------------------------
c     n: number of odes
c     t: time independent variable
c     y: solution (y(1)=x, y(2)=p)
c     f: derivative of solution
c-----------------------------------------------------------------------
      integer n
      parameter (n = 2)
      real t
      real y(n), f(n)

c-----------------------------------------------------------------------
c     evaluate derivatives
c-----------------------------------------------------------------------
      f(1) = y(2)/m
      f(2) = v0 + v*cos(w*t + theta)

      return

      end

c=======================================================================
c     this is a function that monitors collisions with the infinite
c     wall in the driven triangle well
c=======================================================================
      real function g(t, y)

      implicit none

c-----------------------------------------------------------------------
c     common commmunication with main program
c-----------------------------------------------------------------------
      include 'fcn.inc'

      integer n
      parameter (n = 2)
      real t
      real y(n)

      g = y(1) - l

      return

      end

c=======================================================================
c This is a common block include file for use with the routine
c twcmap.f.  The parameters are:
c
c V: strength of driving field
c V0: strength of triangle well
c w: frequency of driving field
c theta: phase "
c Per: period "
c m: mass of particle
c L: position of infinite wall
c=======================================================================
      integer i
      real*8 V, V0, w, theta, L, Per, m

      common /com_fcn/
     & i,
     & V,V0,w,theta,
     & L,Per,m
```

7.2 Calculating Energy Eigenvalues of Triangle Well

```
c=======================================================================
c     History: newt1.f
c
c     This program computes eigenvalues for the triangular well:
c
c     H0 = p**2/(2*m) -V0*x
c
c     The routine uses a Newton's method approach to solve for zeros
c     of the Airy function, and uses these to calculate eigenvalues
c     for n = 1,...,maxst.
c=======================================================================
      program triwev

      implicit none

c-----------------------------------------------------------------------
c     Argument parsing variables
c-----------------------------------------------------------------------
      integer iargc
      real*8     r8arg, r8_never
      parameter (r8_never = -1.0d-60)

      real*8     v0, l, tol
      real*8     default_tol
      parameter (default_tol = 1.0d-12)

c-----------------------------------------------------------------------
c     Filename where eigenvalues will be written
c-----------------------------------------------------------------------
      character*(*) Efile
      parameter (Efile = 'trianglE')

      integer getu, ufrom

c-----------------------------------------------------------------------
c     Hard-code parameters hbar and m
c-----------------------------------------------------------------------
      real*8     hbar, m
      parameter (hbar = 1.0d0)
      parameter (m = 1.0d0)

c-----------------------------------------------------------------------
c     maxst: maximum state for which E will be computed
c     E: energy eigenvalue
c-----------------------------------------------------------------------
      integer ist, maxst
      parameter (maxst = 1 000)

      real*8     e, s, pi

c-----------------------------------------------------------------------
c     Newton variables:
c     res: residual values
c     x: current trial value
c     dx: correction to trial value
c     iter: current iteration number
c     mxiter: maximum number of Newton iterations
c     nrm2_: 1-2 norm of quantity _ (absolute value for 1-d)
c-----------------------------------------------------------------------
      real*8     fprime, res, x, dx

      integer mxiter
      parameter (mxiter = 10)

      integer iter
      real*8     nrm2res, nrm2dx, nrm2x
      real*8     drelabs

c-----------------------------------------------------------------------
c     Airy function and derivative from imsl library
c-----------------------------------------------------------------------
      real*8     dai, daid
      external dai, daid

c-----------------------------------------------------------------------
c     Argument parsing
c-----------------------------------------------------------------------
      if (iargc() .lt. 2) go to 900

      v0 = r8arg(1, r8_never)
      l  = r8arg(2, r8_never)
      if (v0 .eq. r8_never .or. l .eq. r8_never) go to 900

      tol = r8arg(3, default_tol)
      if (tol .le. 0.0d0) tol = default_tol

      pi = 4.0d0*atan(1.0d0)

c-----------------------------------------------------------------------
c     Get available unit number and open file fname
```

```
c-----------------------------------------------------------------------
      ufrom = getu()
      open(ufrom,file=Efile,status='new')

c-----------------------------------------------------------------------
c     Loop over states 1 to maxst.  Each pass calculates the energy
c     eigenvalue for the state n = ist
c-----------------------------------------------------------------------
      do ist = 1, maxst

c-----------------------------------------------------------------------
c     Initialize value for zero of airy function.  We use a power
c     series approximation to the zeros (see Handbook of Math. Func.)
c-----------------------------------------------------------------------
         s = 3.0d0*pi*(4.0d0*dble(ist)-1.0d0)/8.0d0

         x = -s**(2.0d0/3.0d0)*(1.0d0 + (5.0d0/(4.8d1*s**2)) -
     &        (5.0d0/(3.6d1*s**4)) + (7.7125d4/(8.2944d4*s**6))
     &        - (1.08056875d8/(6.967296d6*s**8)) +
     &        (1.62375596875d11/(3.34430208d8*s**10)))

         write(0,*) 'Newton iteration for state n = ',ist
         write(0,*) 'Iter       x            log10(dx) '//
     &        ' log10(res)'

c-----------------------------------------------------------------------
c     Begin Newton loop for state n = ist
c-----------------------------------------------------------------------
         do iter = 1, mxiter

c-----------------------------------------------------------------------
c     Evaluate residual and correction
c-----------------------------------------------------------------------
            res = dai(x)
            nrm2res = abs(res)
            fprime = daid(x)
            dx = res/fprime
            nrm2dx = abs(dx)
            nrm2x = abs(x)

c-----------------------------------------------------------------------
c     Update solution value
c-----------------------------------------------------------------------
            x = x - dx

c-----------------------------------------------------------------------
c     Tracing of iterations
c-----------------------------------------------------------------------
            write(0,1000) iter, x, log10(max(nrm2dx,1.0d-60)),
     &           log10(max(nrm2res,1.0d-60))
 1000       format(i2, 1p, e24.16, 2f8.2)

c-----------------------------------------------------------------------
c     Check for convergence
c-----------------------------------------------------------------------
            if (drelabs(nrm2dx,nrm2x,1.0d-6) .lt. tol) go to 100

         end do

c-----------------------------------------------------------------------
c     No convergence exit
c-----------------------------------------------------------------------
         write(0,*) 'Computation terminated at n = ',ist
         write(0,*) 'No convergence after ',mxiter,' iterations.'
         write(0,*) tol
         close(ufrom)
         stop


c-----------------------------------------------------------------------
c     Convergence exit
c-----------------------------------------------------------------------
 100     continue

c-----------------------------------------------------------------------
c     Use solution of Ai(x)=0 to calculate energy eigenvalue
c-----------------------------------------------------------------------
         e = -((v0**2*hbar**2/(2*m))**(1.0d0/3.0d0))*x - v0*l

         write(ufrom,*) ist, e

      end do

c-----------------------------------------------------------------------
c     Close file fname
c-----------------------------------------------------------------------
      close(ufrom)

      stop

c-----------------------------------------------------------------------
c     Usage message
c-----------------------------------------------------------------------
 900  continue
      write(0,*) 'usage: triwev <V0> <L> [<tol>]'
      write(0,*)
      write(0,*) 'writes energies to ',Efile

      stop

      end



c=======================================================================
c     drelabs: function for 'relativizing' quantity being
c     monitored for detection of convergence (by Matt Choptuik)
c=======================================================================
      real*8     function drelabs(dx,x,xfloor)

         implicit none

         real*8     dx, x, xfloor

         if (abs(x) .lt. abs(xfloor)) then
            drelabs = abs(dx)
         else
            drelabs = abs(dx/x)
         end if

         return

      end
```

7.3 Calculating Acceleration Time Series

```
c=======================================================================
c     History: sode.f (Matt Choptuik)
c
c     accel:  This program calulates the expectation value of the
c     acceleration for a quantum system of the form:
c
c     H = H0 - V*x*cos(w*t + theta)
c
c     where H0 is a time-independent, integrable Hamiltonian.  The
c     calculation proceeds by computing decomposing the time-dep.
c     wavefunction into a basis of eigenstates of H0 and computing
c     the coefficients of this decomposition as a function of time.
c
c     For a particular system the subroutines 'dipole' and 'getE'
c     must be set up to calculate or read the proper dipole matrix
c     elements and energy eigenvalues for that particular H0.  The
c     common block in 'fcn.inc' will also have to be modified to
c     contain the correct parameters for the specific H0.
c
c     The output of this program is given as a list of time values
c     and the corresponding value of the (real part) of the expectation
c     value of the acceleration written to standard out.  The imaginary
c     part of the accel. (should be 0) and the total probability
c     (should be 1) are written to standard error.
c
c     A Fourier transform of the output of this program gives the
c     radiation spectrum of the system.  The maximum harmonic that
c     can be seen will be determined by the command-line
c     argument, hmax, used with accel.
c=======================================================================
      program accel

      implicit none

c-----------------------------------------------------------------------
c     Common block for communication with subroutines.
c-----------------------------------------------------------------------
      include 'fcn.inc'

c-----------------------------------------------------------------------
c     Argument parsing variables
c-----------------------------------------------------------------------
      integer iargc, i4arg
      real*8     r8arg
      real*8     r8_never
      parameter (r8_never = -1.0d-60)

c-----------------------------------------------------------------------
c     minst: minimum state in basis
c     maxst: maximum "
c     nst : number of states in basis
c     E: vector containing energy eigenvalues of H0
c     x: array containing dipole matrix elements of H0
c-----------------------------------------------------------------------
      integer minst, maxst
      parameter (minst = 1, maxst = 400)
      integer nst
```

```
      parameter (nst = maxst - minst + 1)
      real*8     e(minst:maxst)
      real*8     x(minst:maxst,minst:maxst)
c-----------------------------------------------------------------------
c     nc: Number of coefficients in the expansion of the wavefunction
c            = 2*Number of states in basis
c     ist,jst,kst: state indices for do loops
c     neq: vector containing system dimensions (used in fcn.f)
c     ic: coefficient index for do loops
c-----------------------------------------------------------------------
      integer nc, ist, jst, kst, ic, jc
      parameter (nc = 2*nst)
      integer neq(4)

c-----------------------------------------------------------------------
c     Initial state at t = 0
c-----------------------------------------------------------------------
      integer inst
      parameter (inst = 60)

c-----------------------------------------------------------------------
c     Maximum number of output times
c-----------------------------------------------------------------------
      integer maxout, p2max, p2
      parameter (p2max = 14)
      parameter (maxout = 2**p2max)

c-----------------------------------------------------------------------
c     ncyc: Number of field cycles over which to compute accel
c     hmax: Maximum harmonic seen in radiation spectrum
c     tout: Vector holding output times
c     ntout: Total number of output times
c-----------------------------------------------------------------------
      integer ncyc, hmax
      parameter (ncyc = 256)
      real*8     tout(maxout), work(maxout)
      integer ntout, itout

c-----------------------------------------------------------------------
c     lsoda tolerance
c-----------------------------------------------------------------------
      real*8     tol, default_tol
      parameter (default_tol = 1.0d-8)

c-----------------------------------------------------------------------
c     phi: phase angle of initial state
c     pi: 3.14...
c-----------------------------------------------------------------------
      real*8      phi, pi
      parameter (phi = 0.0d0)

c-----------------------------------------------------------------------
c     Logical variables to determine tracing
c-----------------------------------------------------------------------
      logical ltrace
      parameter (ltrace = .true.)

      logical lsodatrace
      parameter (lsodatrace = .false.)

      logical strace
      parameter (strace = .false.)

      logical ptrace
      parameter (ptrace = .true.)

      logical vartrace
      parameter (vartrace = .true.)

c-----------------------------------------------------------------------
c     Variables for output of wavefunction, if (ptrace)
c
c     fname: Name of file in which the data for the wavefunction will
c            be stored (it will have a .hdf ending added)
c     pst: array storing occupation probabilities for energy states
c     npout: # of times for which pst will be written to output
c            (npout should always be a power of 2)
c     pint: pst written every pint'th step
c-----------------------------------------------------------------------
      character*(*) fname
      parameter (fname = 'wvfcn')

      integer pint, npout
      parameter (npout = 2 048)

      real*8     pst(nst)
      integer shape, rank
      real*8 bbox(2)

c-----------------------------------------------------------------------
c     varfile: file to which variances will be written
c     navg,n2avg: 1st and 2nd moments of distribution
c     var: variance of distribution in n (energy level index)
c-----------------------------------------------------------------------
      character*(*) varfile
      parameter (varfile = 'varn.dat')

      integer ufrom, getu
      real*8 navg, var

c-----------------------------------------------------------------------
c     Efile: file from which energy eigenvalues will be read
c-----------------------------------------------------------------------
      character*(*) Efile
      parameter (Efile = 'trianglE')

c-----------------------------------------------------------------------
c     lsoda variables
c-----------------------------------------------------------------------
      external fcn, jac

      real*8     y(nst*(nst + 3))
      real*8     yprime(nc)
      real*8     tbgn, tend
      integer itol
      real*8     rtol, atol
      integer itask, istate, iopt
      integer lrw
      parameter (lrw = 22 + 9*nc + nc**2)
      real*8     rwork(lrw)
      integer liw
      parameter (liw = 20 + nc)
      integer iwork(liw)
      integer jt

c-----------------------------------------------------------------------
c     tdacc(j,i): coefficient of time dependent term in acceleration
c                 'matrix element' (multiply by V*z(t)*cos(wt+theta))
c     accti(j,i): time independent term "
c     accre: real part of the expectation value of acceleration
c     accim: imaginary part (should be 0)
c     prob: total probability in the wavefunction (should be 1)
c-----------------------------------------------------------------------
      real*8     tdacc(minst:maxst,minst:maxst)
      real*8     accti(minst:maxst,minst:maxst)
      real*8     accre, accim
      real*8     prob

c-----------------------------------------------------------------------
c     Set parameters defiined in common block
c     V0: strength of triangle well
c     L: width of triangle well
c     m: mass of quantum particle
c     hbar: obvious
c     theta: initial phase of radiation field
c-----------------------------------------------------------------------
      v0 = 4.0d-1
      l = 6.0d2
      m = 1.0d0
      hbar = 1.0d0
      theta = 0.0d0

c-----------------------------------------------------------------------
c     Set parameters for output of wavefunction if (ptrace)
c-----------------------------------------------------------------------
      bbox(1) = minst
      bbox(2) = maxst
      rank = 1
      shape = nst

c-----------------------------------------------------------------------
c     Argument parsing and definition of pi.
c     w: frequency of the radiation field
c     V: strength of the radiation field
c     hmax: maximum harmonic in radiation spectrum
c     tocyc: # of cycles in turn-on
c     tol: optional tolerance for lsoda
c-----------------------------------------------------------------------
      if (iargc() .lt. 4) go to 900

      w = r8arg(1, r8_never)
      v = r8arg(2, r8_never)
      if (w .eq. r8_never .or. v .eq. r8_never) go to 900

      hmax = i4arg(3, -1)
      tocyc = i4arg(4, -1)
      if (hmax .lt. 0 .or. tocyc .lt. 0) go to 900

      tol = r8arg(5, default_tol)

      pi = 4.0d0*atan(1.0d0)

c-----------------------------------------------------------------------
c     Set values for totime, ntout.
```

```
c     totime: total time for turn-on of laser field
c-----------------------------------------------------------------------
      totime = 2.0d0*pi*tocyc/w

      ntout = 2*hmax*ncyc

      if (ntout .gt. maxout) then
         write(0,*) 'Number of output times exceeds internal'//
     &        ' storage. Use lower hmax or change source code.'
         stop
      end if

c-----------------------------------------------------------------------
c     Check that ntout .gt. npout and set value of pint
c-----------------------------------------------------------------------
      if (ntout .lt. npout) then
         write(0,*) 'ntout .lt. npout! (modify npout in source code)'
         stop
      else
         pint = ntout/npout
      end if

c-----------------------------------------------------------------------
c     Check to see if ntout is a power of 2
c-----------------------------------------------------------------------
      do p2 = 1, p2max
         if (ntout .eq. 2**p2) go to 10
      end do

      write(0,*) 'Number of output times should be a power of 2!'
      write(0,*) 'Make sure that hmax and ncycle are powers of 2.'
      stop

 10   continue

c-----------------------------------------------------------------------
c     Set values in neq(4) for use with subroutine 'fcn'.
c-----------------------------------------------------------------------
      neq(1) = nc
      neq(2) = minst
      neq(3) = maxst
      neq(4) = nst

c-----------------------------------------------------------------------
c     Calculate and store output times
c-----------------------------------------------------------------------
      do itout = 1, ntout
         tout(itout) = 0.0d0 + (itout - 1)*pi/(w*hmax)
      end do

c-----------------------------------------------------------------------
c     Read in vector of energy eigenvalues using subroutine getE
c-----------------------------------------------------------------------
      call getE(Efile,minst,maxst,E)

c-----------------------------------------------------------------------
c     Read in dipole matrix elements using subroutine dipole
c-----------------------------------------------------------------------
      call dipole(minst,maxst,E,x)

c-----------------------------------------------------------------------
c     Tracing of getE and dipole subroutine calls
c-----------------------------------------------------------------------
      if (strace) then
         write(0,*) 'minst = ',minst,', maxst = ',maxst
         do ist = minst, maxst
            write(0,*) 'E(',ist,') = ',E(ist)
         end do
         write (0,*) 'dipole elements'
         do ist = minst, maxst
            write(0,450) (x(ist,jst), jst = minst, maxst)
 450        format(1p,5e15.6)
         end do
      end if

c-----------------------------------------------------------------------
c     Calculate accti(j,i) and tdacc(j,i)
c-----------------------------------------------------------------------
      do ist = minst, maxst
         do jst = minst, maxst
            accti(jst,ist) = -((E(jst)-E(ist))**2)*x(jst,ist)/
     &           hbar**2
            tdacc(jst,ist) = 0.0d0

            do kst = minst, maxst
               tdacc(jst,ist) = tdacc(jst,ist) + (2.0d0*e(kst)-
     &              E(ist)-E(jst))*x(jst,kst)*x(kst,ist)/hbar**2
            end do

         end do
      end do
```

```
c-----------------------------------------------------------------------
c     Initialize the coefficients of the wavefunction
c     Note: y(ic) = a(ic+maxst-nst)
c           y(ic+nst) = b(ic+maxst-nst)
c-----------------------------------------------------------------------
      do ic = 1, nc
         y(ic) = 0.0d0
      end do

      y(inst-maxst+nst) = cos(phi)
      y(inst-maxst+2*nst) = sin(phi)

c-----------------------------------------------------------------------
c     Store E and x as elements of the vector y.  This is necessary to
c     pass the values of E and x to the subroutine 'fcn'.
c     y(ic+2*nst) = E(ic+maxst-nst)
c     y(nst*ic+jc+2*nst) = x(jc+maxst-nst,ic+maxst-nst)
c-----------------------------------------------------------------------
      do ic = 1, nst
         y(ic+2*nst) = E(ic+maxst-nst)
      end do

      do ic = 1, nst
         do jc = 1, nst
            y(nst*ic+jc+2*nst) = x(jc+maxst-nst,ic+maxst-nst)
         end do
      end do

c-----------------------------------------------------------------------
c     Calculate accre, accim, and prob at
c     t = tout(1) = 0.
c-----------------------------------------------------------------------
      if (tout(1) .ge. totime) then
         accre = tdacc(inst,inst)*v*cos(theta)
      else
         accre = 0.0d0
      end if

      accim = 0.0d0
      prob = 1.0d0

      if (ltrace) then
         write(0,500) tout(1), prob, accim
 500     format(' ','At step 1, t = ',1pe10.3,', prob = ',1pe12.5,
     &        ', accim = ',1pe12.5)
      end if

      write(*,*) tout(1), accre

c-----------------------------------------------------------------------
c     Tracing output of probabilities in each state to file fname
c-----------------------------------------------------------------------
      if (ptrace) then
         do ic = 1, nst
            pst(ic+maxst-nst) = y(ic)*y(ic) + y(ic+nst)*y(ic+nst)
         end do
         call gft_write_bbox(fname,tout(1),shape,rank,bbox,pst)
      end if

c-----------------------------------------------------------------------
c     Calculate and write var at initial time
c-----------------------------------------------------------------------
      if (vartrace) then
         ufrom = getu()
         open(ufrom,file=varfile,status='new')
         navg = 0.0d0
         var = 0.0d0
         do ic = 1, nst
            navg = navg + (ic+maxst-nst)*(y(ic)*y(ic) +
     &           y(ic+nst)*y(ic+nst))
         end do
         write(0,*) 'At t = ',tout(1),' navg = ',navg
         do ic = 1, nst
            var = var + (real(ic+maxst-nst)-navg)*
     &           (real(ic+maxst-nst)-navg)*(y(ic)*y(ic) +
     &           y(ic+nst)*y(ic+nst))
         end do
         write(ufrom,*) tout(1)*w/(2.0d0*pi), var
      end if

c-----------------------------------------------------------------------
c     Set lsoda parameters
c-----------------------------------------------------------------------
      itol = 1
      rtol = tol
      atol = tol
      itask = 1
      iopt = 0
      jt = 2

c=======================================================================
c     BEGIN DO LOOP OVER TIME VALUES
```

```
c=======================================================================
      do itout = 2, ntout
         istate = 1

c-----------------------------------------------------------------------
c     Need these temporaries since lsoda overwrites tend ...
c-----------------------------------------------------------------------
         tbgn = tout(itout - 1)
         tend = tout(itout)

c-----------------------------------------------------------------------
c     Call ODE integrator lsoda.  Returns solution at t = tend in
c     y(neq) (i.e. y is overwritten).
c-----------------------------------------------------------------------
         call lsoda(fcn,neq,y,tbgn,tend,itol,rtol,atol,itask,
     &        istate,iopt,rwork,lrw,iwork,liw,jac,jt)

         if (lsodatrace) then
            write(0,1000) itout, ntout, tout(itout-1), tout(itout)
 1000       format(' ','accel: Step ',i4,' of ',i4,' t = ',
     &           1pe10.3,' .. ', 1pe10.3)
            write(0,*) 'accel: lsoda returns ',istate
         end if

c-----------------------------------------------------------------------
c     Error message if lsoda fails
c-----------------------------------------------------------------------
         if (istate .lt. 0) then
            write(0,1500) istate, itout, ntout, tout(itout-1),
     &           tout(itout)
 1500       format(/' ','accel: Error retrun ',i2,
     &           ' from integrator lsoda.'/
     &           '   At output time ',i5,' of ',i5/
     &           '   Interval ',1pe10.3,' .. ',1pe10.3/)
            stop
         end if

c-----------------------------------------------------------------------
c     Tracing output of probabilities p(ist) to file fname
c-----------------------------------------------------------------------
         if (ptrace .and. (mod(ntout,pint) .eq. 0)) then
            do ic = 1, nst
               pst(ic+maxst-nst) = y(ic)*y(ic) + y(ic+nst)*y(ic+nst)
            end do
            call gft_write_bbox(fname,tout(itout),shape,rank,bbox,pst)
         end if

c-----------------------------------------------------------------------
c     Output of variance to file varfile
c-----------------------------------------------------------------------
         if (vartrace) then
            navg = 0.0d0
            var = 0.0d0
            do ic = 1, nst
               navg = navg + (ic+maxst-nst)*(y(ic)*y(ic) +
     &              y(ic+nst)*y(ic+nst))
            end do
            write(0,*) 'At t = ',tout(itout),' navg = ',navg
            do ic = 1, nst
               var = var + (real(ic+maxst-nst)-navg)*
     &              (real(ic+maxst-nst)-navg)*(y(ic)*y(ic) +
     &              y(ic+nst)*y(ic+nst))
            end do
            write(ufrom,*) tout(itout)*w/(2.0d0*pi), var
         end if

c-----------------------------------------------------------------------
c     Calculate total probability at time t = tout(itout)
c-----------------------------------------------------------------------
         if (ltrace) then
            prob = 0.0d0

            do ic = 1, nc
               prob = prob + y(ic)*y(ic)
            end do
         end if

c-----------------------------------------------------------------------
c     Calculate accre and accim at time t = tout(itout)
c-----------------------------------------------------------------------
         accre = 0.0d0
         accim = 0.0d0

         if  (tout(itout) .lt. totime) then
            do ic = 1, nst
               do jc = 1, nst
                  accre = accre +
     &                 (y(jc)*y(ic) + y(jc+nst)*y(ic+nst))*
     &                 (accti(jc+maxst-nst,ic+maxst-nst)
     &                 + tdacc(jc+maxst-nst,ic+maxst-nst)*v*
     &                 (sin(w*tout(itout)/(4.0d0*tocyc))**2)*
     &                 cos(w*tout(itout)+theta))
                  accim = accim +
     &                 (y(jc)*y(ic+nst) - y(jc+nst)*y(ic))*
     &                 (accti(jc+maxst-nst,ic+maxst-nst)
     &                 + tdacc(jc+maxst-nst,ic+maxst-nst)*v*
     &                 (sin(w*tout(itout)/(4.0d0*tocyc))**2)*
     &                 cos(w*tout(itout)+theta))
               end do
            end do
         else
            do ic = 1, nst
               do jc = 1, nst
                  accre = accre +
     &                 (y(jc)*y(ic) + y(jc+nst)*y(ic+nst))*
     &                 (accti(jc+maxst-nst,ic+maxst-nst)
     &                 + tdacc(jc+maxst-nst,ic+maxst-nst)*v*
     &                 cos(w*tout(itout)+theta))
                  accim = accim +
     &                 (y(jc)*y(ic+nst) - y(jc+nst)*y(ic))*
     &                 (accti(jc+maxst-nst,ic+maxst-nst)
     &                 + tdacc(jc+maxst-nst,ic+maxst-nst)*v*
     &                 cos(w*tout(itout)+theta))
               end do
            end do
         end if

c-----------------------------------------------------------------------
c     Tracing output: prob and accim
c-----------------------------------------------------------------------
         if (ltrace) then

            write(0,2000) itout, tout(itout), prob,
     &           accim
 2000       format(' ','At step ',i5,', t = ',1pe10.3,', prob = ',
     &           1pe12.5,', accim = ',1pe12.5)

         end if

c-----------------------------------------------------------------------
c     Write time series to standard output and stop
c-----------------------------------------------------------------------
         write(*,*) tout(itout), accre

c-----------------------------------------------------------------------
c     END DO LOOP OVER OUTPUT TIMES
c-----------------------------------------------------------------------
      end do

      if (ptrace) then
         call gft_close_all()
      end if

      if (vartrace) then
         close(ufrom)
      end if

      stop

c-----------------------------------------------------------------------
c     Usage message
c-----------------------------------------------------------------------
 900  continue
      write(0,*) 'usage: accel <w> <V> <hmax> <tocyc> [<tol>]'
      write(0,*)
      write(0,*) 'w: frequency of driving field'
      write(0,*) 'V: strength of driving field'
      write(0,*) 'hmax: maximum harmonic order'
      write(0,*) 'tocyc: # of cycles in turn-on'
      write(0,*) 'tol: optional tolerance for lsoda'
      write(0,*)
      write(0,*) 'Output written to standard out'
      write(0,*) 'Probabilities written to ',fname,' if (ptrace)'
      write(0,*) 'Energies read from ',Efile

      stop

      end

c=======================================================================
c     This is a subroutine that calculates (or reads from a file)
c     the energy eigenvalues of a conservative quantum system for
c     use with the 'accel' program (accel.f).  This routine is
c     currently set up to calculate energy eigenvalues for the
c     triangle well:
c
c     H = p**2/(2*m) - V0*x,        where x < L (infinite wall at L)
c
c     Note: This program is currently set to read from the file
c     'trianglE'.  This file is a list of eigenvalues created by the
c     program 'triwev' (triwev.f) using parameters: V0=0.4 L=600.
c=======================================================================
      subroutine gete(fname,minst,maxst,e)
```

```
      implicit none

c-----------------------------------------------------------------------
c     fname: File containing energy eigenvalues (not used currently)
c     minst: minimum state whose energy is calculated
c     maxst: maximum state whose energy is calculated
c     E: vector storing energy eigenvalues
c     mxmxst: Largest value of maxst for which eigenvalues have
c             been computed
c-----------------------------------------------------------------------
      character*(*) fname
      integer ist, jst
      real*8   ejst
      integer minst, maxst
      real*8     e(minst:maxst)
      integer mxmxst
      parameter (mxmxst = 1000)
      integer getu, ufrom
      integer indlnb, rc

c-----------------------------------------------------------------------
c     Check if maxst .gt. mxmxst
c-----------------------------------------------------------------------
      if (maxst .gt. mxmxst) then
         write(0,*) 'File ',fname(1:indlnb(fname)),' only '//
     &          'contains ',mxmxst,' eigenvalues.'
         write(0,*) 'Use smaller maxst or recompute eigenvalues'
         return
      end if

c-----------------------------------------------------------------------
c     Get available unit number
c-----------------------------------------------------------------------
      ufrom = getu()

c-----------------------------------------------------------------------
c     Open file fname for formatted I/O
c-----------------------------------------------------------------------
      open(ufrom,file=fname(1:indlnb(fname)),
     &       form='formatted',status='old',iostat=rc)

      if (rc .ne. 0) then
c-----------------------------------------------------------------------
c     Couldn't open file, print error message and return
c-----------------------------------------------------------------------
         write(0,*) 'getE: Error opening ',fname(1:indlnb(fname))
         return
      end if

c-----------------------------------------------------------------------
c     Input eigenvalues into vector E, making sure to start at minst
c     and stop at maxst
c-----------------------------------------------------------------------
      do ist = 1, maxst
         read(ufrom,*,iostat=rc) jst, ejst
         if (rc .eq. 0) then
            if (jst .ge. minst) then
               if (jst .le. maxst) then
                  e(jst) = ejst
               end if
            end if
         else
            write(0,*) 'Error reading from ',
     &           fname(1:indlnb(fname))
            write(0,*) fname(1:indlnb(fname)),' may not have '//
     &           'enough eigenvalues (needs ',maxst,')'
            return
         end if
      end do

      return

      end

c=======================================================================
c     This subroutine computes dipole matrix elements for 1-D
c     quantum systems.  It is currently set up for the system
c
c     H0 = p**2/(2*m) - V0*x,  x < L (infinite wall at L)
c
c     This routine is used in the 'accel' program (accel.f)
c=======================================================================
      subroutine dipole(minst,maxst,E,x)

         implicit none

         integer ist, jst

c-----------------------------------------------------------------------
c     minst: minimum state in basis
c     maxst: maximum "
c     E: vector containing energy eigenvalues
c     x: array containing dipole matrix elements
c-----------------------------------------------------------------------
         integer minst, maxst
         real*8 E(minst: maxst)
         real*8 x(minst:maxst,minst:maxst)

c-----------------------------------------------------------------------
c     Common block for communication with accel.f, getE.f, fcn.f
c-----------------------------------------------------------------------
         include 'fcn.inc'

c-----------------------------------------------------------------------
c     Compute dipole matrix elements
c-----------------------------------------------------------------------
         do ist = minst,  maxst
            do jst = minst, maxst
               if (ist .eq. jst) then
                  x(jst,ist) = -2.0d0*E(ist)/(3.0d0*V0)
               else
                  x(jst,ist) = -V0/(m*(E(jst)-E(ist))**2)
               end if
            end do
         end do

         return

      end

c=======================================================================
c     This is a subroutine that is used with the 'accel' program
c     (accel.f).  It is used to evaluate the derivatives of the
c     coefficients in the expansion of a quantum state with
c     Hamiltonian:
c
c     H = H0 - V*x*cos(w*t + theta)
c
c     where H0 is a time-independent, integrable Hamiltonian.
c
c     The wavefunction is expanded in the energy basis of H0.
c=======================================================================
      subroutine fcn(neq, t, y, yprime)

         implicit none

c-----------------------------------------------------------------------
c     Common block for communication with accel.f, getE.f, dipole.f
c-----------------------------------------------------------------------
         include 'fcn.inc'

c-----------------------------------------------------------------------
c     Arguments of subroutine and indices
c     Note: dimension of y is nst*(nst+3) (nst = dims(4))
c           dimension of yprime is neq = dims(1)
c-----------------------------------------------------------------------
         integer ic, jc
         integer minst, maxst, nst
         real*8 t
         integer neq
         real*8 y, yprime

         dimension neq(4)
         dimension y(1), yprime(1)

c-----------------------------------------------------------------------
c     Temporaries for dimension of system
c-----------------------------------------------------------------------
         minst = neq(2)
         maxst = neq(3)
         nst = neq(4)

c-----------------------------------------------------------------------
c     Define derivatives
c     Note: y(ic) = a(ic+maxst-nst)
c           y(ic+nst) = b(ic+maxst-nst)
c           y(ic+2*nst) = E(ic+maxst-nst)
c           y(nst*ic+jc+2*nst) = x(jc+maxst-nst,ic+maxst-nst)
c
c     Note: we will replace x(j,i) with x(i,j) in the calculations
c     below for better vectorization.  We can do this as long as
c     the x(j,i) are real.
c-----------------------------------------------------------------------
         if (t .lt. totime) then

            do ic = 1, nst
               yprime(ic) = y(ic+2*nst)*y(ic+nst)/hbar

               yprime(ic+nst) = - y(ic+2*nst)*y(ic)/hbar

               do jc = 1, nst
                  yprime(ic) = yprime(ic) -
     &                 V*(sin(w*t/(4.0d0*tocyc))**2)*
```

```
     &                 cos(w*t+theta)*
     &                 y(nst*ic+jc+2*nst)*y(jc+nst)/hbar

              yprime(ic+nst) = yprime(ic+nst)
     &                 +V*(sin(w*t/(4.0d0*tocyc))**2)*
     &                 cos(w*t+theta)*
     &                 y(nst*ic+jc+2*nst)*y(jc)/hbar
            end do
         end do

      else

         do ic = 1, nst
            yprime(ic) = y(ic+2*nst)*y(ic+nst)/hbar

            yprime(ic+nst) = - y(ic+2*nst)*y(ic)/hbar

            do jc = 1, nst
               yprime(ic) = yprime(ic) -
     &                 V*cos(w*t+theta)*y(nst*ic+jc+2*nst)
     &                 *y(jc+nst)/hbar

               yprime(ic+nst) = yprime(ic+nst) +
     &                 V*cos(w*t+theta)*y(nst*ic+jc+2*nst)
     &                 *y(jc)/hbar
            end do

         end do
      end if

      return

      end

c=======================================================================
c     Optional Jacobian for lsoda (see accel.f)
c=======================================================================
      subroutine jac(neq,t,y,ml,mu,pd,nrowpd)

         implicit none

         include 'fcn.inc'

         integer ic, jc
         integer minst, maxst, nst
         real*8 t
         integer neq(4)
         real*8 y, pd
         integer ml, mu, nrowpd

         dimension y(1), pd(nrowpd, 1)

c----------------------------------------------------------------------
c     Set values of minst, maxst, nst from array neq
c-----------------------------------------------------------------------
         minst = neq(2)
         maxst = neq(3)
         nst = neq(4)

c-----------------------------------------------------------------------
c     Calculate Jacobian.  Note:  this routine only works because
c     the x(j,i) are real and thus x(j,i) = x(i,j).  If the x(j,i) are
c     complex then the entier "accel" program must be altered.
c-----------------------------------------------------------------------
         if (t .lt. totime) then

c-----------------------------------------------------------------------
c     If we are still in turn-on... calculate bottom left of jac
c-----------------------------------------------------------------------
            do ic = 1, nst
               do jc = nst+1, 2*nst
                  if (jc .eq. ic+nst) then
                     pd(jc,ic) = -y(ic+2*nst)/hbar + V*(sin(w*t/
     &                    (4.0d0*tocyc))**2)*cos(w*t+theta)*
     &                    y((nst+1)*ic+2*nst)/hbar
                  else
                     pd(jc,ic) = V*(sin(w*t/(4.0d0*tocyc))**2)*
     &                    cos(w*t+theta)*y(nst*ic+jc+nst)/hbar
                  end if
               end do
            end do

c-----------------------------------------------------------------------
c     and calculate top right of jac
c-----------------------------------------------------------------------
            do ic = nst+1, 2*nst
               do jc = 1, nst
                  pd(jc,ic) = -pd(ic,jc)
               end do
            end do

         else

c-----------------------------------------------------------------------
c     If we are out of turn-on... calculate bottom left of jac
c-----------------------------------------------------------------------
            do ic = 1, nst
               do jc = nst+1, 2*nst
                  if (jc .eq. ic+nst) then
                     pd(jc,ic) = -y(ic+2*nst)/hbar + V*
     &                    cos(w*t+theta)*y((nst+1)*ic+2*nst)/hbar
                  else
                     pd(jc,ic) = V*cos(w*t+theta)*
     &                    y(nst*ic+jc+nst)/hbar
                  end if
               end do
            end do

c-----------------------------------------------------------------------
c     and calculate top right of jac
c-----------------------------------------------------------------------
            do ic = nst+1, 2*nst
               do jc = 1, nst
                  pd(jc,ic) = -pd(ic,jc)
               end do
            end do
         end if

         return

      end

c=======================================================================
c Common block for use with 'accel'.  Shares values of varibles
c among accel.f, getE.f, dipole.f, and fcn.f.  The variables
c are:
c
c tocyc: # of cycles in turn-on
c V0: strength of triangle well
c m: mass
c L: width of triangle well
c totime: turn-on time
c V: strength of driving field
c w: frequency of "
c theta: initial phase of "
c
c=======================================================================
      integer
     & tocyc

      real*8
     & V0,hbar,m,L,
     & totime,V,w,theta

      common / com_fcn /
     & tocyc,
     & V0,  hbar,m,L,
     & totime,V,w,theta
```

7.4 Computing Fourier Transform of Acceleration

```
c=======================================================================
c     powspec: This routine reads in a times series of acceleration
c     expectation values and calculates a power spectrum.  Two
c     routines from Numerical Recipes in Fortran are used (realft and
c     four1).  This program is designed to compute spectra for systems
c     of the form H = H0 + V(t)*x*cos(w0*t+theta).  The power spectrum
c     is produced as a function of w/w0.  The maximum value of w/w0
c     for which the power spectrum is calculated is given by
c
c     hmax = N/(2*ncyc), where N is the number of points in the
c     time series and ncyc is the number of cycles of w0 in the
c     time series.
c=======================================================================
      program powspec

      implicit none

c-----------------------------------------------------------------------
c     argument parsing variables
c-----------------------------------------------------------------------
      integer iargc, i4arg
      real*8     r8arg, r8_never
      parameter (r8_never = -1.0d-60)

c-----------------------------------------------------------------------
c     maxn: maximum number of points in the time series
c-----------------------------------------------------------------------
      integer maxn, n, i, p2
      parameter (maxn = 16 384)
```

```
c-----------------------------------------------------------------------
c     vectors for storing times, accel, and a modified form of accel
c     that will be used in the fourier transform
c-----------------------------------------------------------------------
      real*8     data(maxn)
      real*8     time(maxn), accel(maxn)

c-----------------------------------------------------------------------
c     w, hmax, ncyc as defined above
c     ho: harmonic order
c     pow: power at a given harmonic order
c-----------------------------------------------------------------------
      real*8     w, hmax
      integer ncyc
      real*8     ho, pow, pi

c-----------------------------------------------------------------------
c     argument parsing: w, ncyc
c-----------------------------------------------------------------------
      if (iargc() .lt. 2) go to 900

      w = r8arg(1,r8_never)
      if (w .eq. r8_never) go to 900

      ncyc = i4arg(2, -1)
      if (ncyc .lt. 0) go to 900

c-----------------------------------------------------------------------
c     read in the acceleration time series
c-----------------------------------------------------------------------
      call rdacc(time,accel,n,maxn)

c-----------------------------------------------------------------------
c     Error message if there is no data read by rdacc
c-----------------------------------------------------------------------
      if (n .lt. 0) then
         write(0,*) 'No values found in standard input.'
         stop
      end if

c-----------------------------------------------------------------------
c     Check to make sure that n is a power of 2 (maximum of 2**14)
c     The fourier transform won't work unless it is
c-----------------------------------------------------------------------
      do p2 = 1, 14
         if (n .eq. 2**p2) go to 1
      end do

      write(0,*) 'Number of data points must be a power of 2!'
      write(0,*) 'Unable to compute Fourier Transform.'
      stop

 1    continue

c-----------------------------------------------------------------------
c     Define pi, hmax
c-----------------------------------------------------------------------
      pi = 4.0d0*atan(1.0d0)

      hmax = dble(n)/(2*dble(ncyc))

c-----------------------------------------------------------------------
c     Modify accel values for fourier transform
c-----------------------------------------------------------------------
      do i = 1, n
         data(i) = accel(i)*(1.0d0-((2.0d0*dble(i)-dble(n-1))/
     &        dble(n+1))**2)
      end do

c-----------------------------------------------------------------------
c     Calculate fourier transform using Num. Rec. routine 'realft'
c-----------------------------------------------------------------------
      call realft(data,n,1)

c-----------------------------------------------------------------------
c     Compute power and write ho, power to standard out
c-----------------------------------------------------------------------
      pow = pi*data(1)**2/(w**2*hmax**2)

      write(*,100) 0.0e0, log10(pow)
 100  format(1p,2e25.16)

      do i = 2, n/2
         ho = dble(i-1)/dble(ncyc)
         pow = pi*(data(2*i-1)**2 + data(2*i)**2)/(w**2*hmax**2)
         write(*,200) ho, log10(pow)
 200     format(1p,2e25.16)
      end do

      stop

 900  continue
```

```
      write(0,*) 'usage: powspec <w> <ncyc>'
      write(0,*)
      write(0,*) 'w: frequency of driving field'
      write(0,*) 'ncyc: # of cycles in time series'
      write(0,*)
      write(0,*) 'reads acceleration data from standard in'
      write(0,*) 'writes spectrum to standard out'

      stop

      end


c=======================================================================
c     History: dvfrom (by Matt Choptuik)
c     this is a subroutine that reads an acceleration time series from
c     standard input and puts the times and accleration values into
c     the vectors time(n) and accel(n).  Note that the dimension n
c     is determined by how many values are in standard input (up to
c     maxn)
c=======================================================================
      subroutine rdacc(time,accel,n,maxn)

         implicit none

         integer n, maxn

         real*8     time(maxn), accel(maxn)

c-----------------------------------------------------------------------
c     Temporaries for reading values of time and accel
c-----------------------------------------------------------------------
         real*8     tn, an

c-----------------------------------------------------------------------
c     return code from read statement
c-----------------------------------------------------------------------
         integer rc, ustdin
         parameter (ustdin = 5)

c-----------------------------------------------------------------------
c     initialize
c-----------------------------------------------------------------------
         n = 0

c-----------------------------------------------------------------------
c     implied do loop to read all values
c-----------------------------------------------------------------------
 100     continue

            read(ustdin,*,iostat=rc,end=200) tn, an
            if (rc .eq. 0) then
               n = n + 1

c-----------------------------------------------------------------------
c     Print error message if there are more than maxn points
c-----------------------------------------------------------------------
               if (n .gt. maxn) then
                  write(0,*) 'rdacc: Read maximum of ',maxn,
     &                 ' from standard input.'
                  n = maxn
                  return
               end if

c-----------------------------------------------------------------------
c     Inject read values into vectors
c-----------------------------------------------------------------------
               time(n) = tn
               accel(n) = an
            end if

            go to 100

c-----------------------------------------------------------------------
c     break out of loop when end-of-file is reached
c-----------------------------------------------------------------------
 200        continue

            return

         end


c=======================================================================
c     This is a double-precision version of the 'realft' routine
c     in Numerical Recipes in Fortran.  It calculates the Fourier
c     transform of a set of n real-valued data points.  The vector
c     containing the data is replaced by a vector containing the
c     fourier components.  It will also calculate the inverse
c     transform of a complex data set if it is the transform of
c     real data.
c=======================================================================
```

```
      subroutine realft(data,n,isign)

         implicit none

         integer isign, n
         real*8     data(n)

         integer i, i1, i2, i3, i4, n2p3
         real*8     c1, c2, h1i, h1r, h2i, h2r
         real*8     theta, wi, wpi, wpr, wr, wtemp

         isign = 1

c-----------------------------------------------------------------------
c     Initialize the recurrence
c-----------------------------------------------------------------------
         theta = 3.141592653589793d0/dble(n/2)
         c1 = 0.5d0

c-----------------------------------------------------------------------
c     Perform the forward fourier transform
c     Note: 'four1' performs the transform
c-----------------------------------------------------------------------
         c2 = -0.5d0
         call four1(data,n/2,+1)

         wpr = -2.0d0*sin(0.5d0*theta)**2
         wpi = sin(theta)
         wr = 1.0d0 + wpr
         wi = wpi
         n2p3 = n + 3

c-----------------------------------------------------------------------
c     Case for i = 1 done separately below
c-----------------------------------------------------------------------
         do i = 2, n/4
            i1 = 2*i - 1
            i2 = i1 + 1
            i3 = n2p3 - i2
            i4 = i3 + 1

c-----------------------------------------------------------------------
c     Separates the two transforms from the combined transform
c-----------------------------------------------------------------------
            h1r = c1*(data(i1)+data(i3))
            h1i = c1*(data(i2)-data(i4))
            h2r = -c2*(data(i2)+data(i4))
            h2i = c2*(data(i1)-data(i3))

c-----------------------------------------------------------------------
c     Separate transforms are recombined to form the true transform
c     of the original data
c-----------------------------------------------------------------------
            data(i1) = h1r+wr*h2r-wi*h2i
            data(i2) = h1i+wr*h2i+wi*h2r
            data(i3) = h1r-wr*h2r+wi*h2i
            data(i4) = -h1i+wr*h2i+wi*h2r

c-----------------------------------------------------------------------
c     The recurrence
c-----------------------------------------------------------------------
            wtemp = wr
            wr = wr*wpr-wi*wpi+wr
            wi = wi*wpr+wtemp*wpi+wi
         end do

c-----------------------------------------------------------------------
c     Squeeze first and last data
c     together to get all values within the original array
c-----------------------------------------------------------------------
         h1r = data(1)
         data(1) = h1r+data(2)
         data(2) = h1r-data(2)

         return

      end


c=======================================================================
c     This routine computes the fourier transform of a complex array
c     of length nn.  It replaces the original data array with the
c     array of complex fourier coefficients.
c
c     Note:  nn must be a power of 2!
c
c     This routine is also from Num. Rec.
c=======================================================================
      subroutine four1(data,nn,isign)

         implicit none

         integer isign, nn
         real*8     data(2*nn)

         integer i, istep, j, m, mmax, n
         real*8     tempi, tempr
         real*8     theta, wi, wpi, wpr, wr, wtemp

         n = 2*nn
         j = 1

c-----------------------------------------------------------------------
c     Perform bit reversal of nn
c-----------------------------------------------------------------------
         do i = 1, n, 2
            if (j .gt. i) then
               tempr = data(j)
               tempi = data(j+1)
               data(j) = data(i)
               data(j+1) = data(i+1)
               data(i) = tempr
               data(i+1) = tempi
            end if

            m = n/2
 1          if ((m .ge. 2) .and. (j .gt. m)) then
               j = j-m
               m = m/2
               goto 1
            end if

            j = j+m
         end do

c-----------------------------------------------------------------------
c     Implement the Danielson-Lanczos algorithm.  The outer loop is
c     executed log2(nn) times
c-----------------------------------------------------------------------
         mmax = 2

 2       if (n .gt. mmax) then
            istep = 2*mmax

c-----------------------------------------------------------------------
c     Initialize for trigonometric recurrence
c-----------------------------------------------------------------------
            theta = 6.28318530717959d0/(isign*mmax)
            wpr = -2.0d0*sin(0.5d0*theta)**2
            wpi = sin(theta)
            wr = 1.0d0
            wi = 0.0d0

c-----------------------------------------------------------------------
c     Two nested inner loops
c-----------------------------------------------------------------------
            do m = 1, mmax, 2

               do i = m, n, istep

c-----------------------------------------------------------------------
c     This is the Danielson-Lanczos formula
c-----------------------------------------------------------------------
                  j = i + mmax
                  tempr = wr*data(j)-wi*data(j+1)
                  tempi = wr*data(j+1)+wi*data(j)
                  data(j) = data(i)-tempr
                  data(j+1) = data(i+1)-tempi
                  data(i) = data(i)+tempr
                  data(i+1) = data(i+1)+tempi
               end do

c-----------------------------------------------------------------------
c     trigonometric recurrence
c-----------------------------------------------------------------------
               wtemp = wr
               wr = wr*wpr-wi*wpi+wr
               wi = wi*wpr+wtemp*wpi+wi
            end do

            mmax = istep

            goto 2

         end if

         return

      end
```

7.5 Calculating Residues and Actions of Periodic Orbits for the Driven Harmonic Oscillator

```
c=====================================================================
c     obitnt: This is a routine for finding the periodic orbits of a
c     driven system.  It uses a shooting method that integrates the
c     equations of motion of a system over a period nT (1<= n<=nmax)
c     combined with a Newton iteration that successively adjusts the
c     initial conditions until the initial and final conditions match.
c
c     This program is currently set up to find the single periodic
c     orbit for the driven harmonic oscillator:
c
c     H = p**2/(2*m) + m*w0**2*x**2/2 + V*x*cos(w*t+theta)
c
c     To modify the routine to find periodic orbits for other systems,
c     one must rewrite the fcn and jac routines in 'fcn.f' and alter
c     the common block in 'fcn.inc'.  Also, any common block
c     parameters must have their values set in this routine.
c=====================================================================
      program orbitnt

      implicit none

c---------------------------------------------------------------------
c     Communication with routines in 'fcn.f'
c---------------------------------------------------------------------
      include 'fcn.inc'

c---------------------------------------------------------------------
c     Argument parsing variables
c---------------------------------------------------------------------
      integer iargc
      real*8 r8arg, r8_never
      parameter (r8_never = -1.0d-60)

c---------------------------------------------------------------------
c     Size of the system (neq = # of odes, npars = # of initial vals.)
c---------------------------------------------------------------------
      integer neq
      parameter (neq = 7)
      integer npars
      parameter (npars = 2)

c---------------------------------------------------------------------
c     Tolerances for lsoda (ltol) and Newton convergence (ntol)
c---------------------------------------------------------------------
      real*8 ltol, default_ltol
      parameter (default_ltol = 1.0d-8)
      real*8 ntol, default_ntol
      parameter (default_ntol = 1.0d-8)

c---------------------------------------------------------------------
c     ivs: array containing initial values (will be altered)
c     J: Jacobian used in Newton step
c     res: residual vector in Newton step
c     y: solution to odes
c---------------------------------------------------------------------
      real*8 ivs(npars), J(npars,npars)
      real*8 res(npars)
      real*8 y(neq)

c---------------------------------------------------------------------
c     S: action along classical path
c     dx,dp: finite differences used in calculation of J above
c     EPS: sqaure root of estimated machine precision
c---------------------------------------------------------------------
      real*8 S
      real*8 dx, dp, EPS

c---------------------------------------------------------------------
c     ipiv: storage needed by dgesv
c     info: return code from dgesv
c     ieq, ipar, iter: indices for eqns, parameters, iterations
c     mxiter: maximum number of Newton iterations at each period
c     nrhs: number of right-hand-sides sent to dgesv = 1
c---------------------------------------------------------------------
      integer ipiv(neq)
      integer info
      integer ieq, ipar, iter
      integer mxiter, nrhs
      parameter (mxiter = 10, nrhs = 1)

c---------------------------------------------------------------------
c     T0: period of driving field
c     iN: index for orbit periods (period = iN*T)
c     maxN: maximum period to be searched
c---------------------------------------------------------------------
      real*8 T0, pi
      integer iN, maxN
      parameter (maxN = 10)
```

```
c---------------------------------------------------------------------
c     x0: permanent storage for inital value of x (not altered)
c     p0: ditto for p
c     R: residue of the periodic orbit
c---------------------------------------------------------------------
      real*8 x0, p0
      real*8 R

c---------------------------------------------------------------------
c     norms used in monitoring convergence of Newton iterations
c---------------------------------------------------------------------
      real*8 nrm2res, nrm2upd, nrm2ivs
      real*8 drelabs, dv12norm

c---------------------------------------------------------------------
c     lsoda variables
c---------------------------------------------------------------------
      external fcn, jac
      real*8 tbgn, tend
      integer itol
      real*8 rtol, atol
      integer itask, istate, iopt
      integer ml, mu
      parameter (ml = 1, mu = 2)
      integer lrw
      parameter (lrw = 20 + 16*neq)
      real*8 rwork(lrw)
      integer liw
      parameter (liw = 20 + neq)
      integer iwork(liw)
      integer jt

c---------------------------------------------------------------------
c     Logical variables for tracing options
c---------------------------------------------------------------------
      logical ltrace
      parameter (ltrace = .true.)

      logical otrace
      parameter (otrace = .true.)

c---------------------------------------------------------------------
c     ofile: file into which orbit coords. will be written
c---------------------------------------------------------------------
      character*(*) ofile
      parameter (ofile = 'orbits')
      integer ufrom, getu

c---------------------------------------------------------------------
c     notpp: number of times orbit coords will be written per period
c     maxout: maximum number of output times
c     tout: array containing output times
c---------------------------------------------------------------------
      integer notpp
      parameter (notpp = 100)
      integer maxout, itout
      parameter (maxout = notpp*maxN)
      real*8 tout(maxout)

c---------------------------------------------------------------------
c     Parse command line arguments
c---------------------------------------------------------------------
      if (iargc() .lt. npars) go to 900
      do ipar = 1, npars
         ivs(ipar) = r8arg(ipar, r8_never)
         if (ivs(ipar) .eq. r8_never) go to 900
      end do

      ltol = r8arg(npars+1, default_ltol)
      ntol = r8arg(npars+2, default_ntol)
      if (ltol .le. 0.0d0) ltol = default_ltol
      if (ntol .le. 0.0d0) ntol = default_ntol

      pi = 4.0d0*atan(1.0d0)

c---------------------------------------------------------------------
c     Set values for common block parameters
c     w: frequency of driving field
c     theta: initial phase of driving field
c     V: strength of driving field
c     m: mass of particle
c     W: harmonic oscillator frequency
c---------------------------------------------------------------------
      w = 4.0d0*pi
      theta = 0.0d0
      m = 1.0d0
      w0 = 4.0d0
      V = 1.0d-1

c---------------------------------------------------------------------
c     Set values of x0, p0, T, EPS
```

```
c-----------------------------------------------------------------------
      x0 = ivs(1)
      p0 = ivs(2)
      T0 = 2.0d0*pi/w
      EPS = sqrt(ltol)

c-----------------------------------------------------------------------
c     Tracing of parameters
c-----------------------------------------------------------------------
      if (ltrace) then
         write(0,*) 'w = ',w,', theta = ',theta,', m = ',m
         write(0,*) 'w0 = ',w0,', V = ',V,', T0 = ',T0
         write(0,*) 'x0 = ',x0,', p0 = ',p0,', EPS = ',EPS
         write(0,*) 'ltol = ',ltol,', ntol = ',ntol
      end if

c-----------------------------------------------------------------------
c     Set lsoda parameters
c-----------------------------------------------------------------------
      itol = 1
      rtol = ltol
      atol = ltol
      itask = 1
      iopt = 0
      jt = 4
      iwork(1) = ml
      iwork(2) = mu

c-----------------------------------------------------------------------
c     Open file ofile
c-----------------------------------------------------------------------
      if (otrace) then
         ufrom = getu()
         open(ufrom,file=ofile,status='new')
      end if

      write(*,*) ' N             R                        S'

c-----------------------------------------------------------------------
c     Begin loop over periods NT
c-----------------------------------------------------------------------
      do iN = 1, maxN
         write(0,*) 'For orbits N = ',iN
         write(0,*)
         write(0,*) '       Iter         x0             p0    '//
     &        '            log10(upd)  log10(res)'
         write(0,*)

c-----------------------------------------------------------------------
c     Reset initial values to original values
c-----------------------------------------------------------------------
         ivs(1) = x0
         ivs(2) = p0

c-----------------------------------------------------------------------
c     Begin Newton loop
c-----------------------------------------------------------------------
         do iter = 1, mxiter
            dx = max(EPS*abs(ivs(1)),1d-8)
            dp = max(EPS*abs(ivs(2)),1d-8)
            y(1) = 0.0d0
            y(2) = ivs(1)
            y(3) = ivs(2)
            y(4) = ivs(1)
            y(5) = ivs(2) + dp

c-----------------------------------------------------------------------
c     Trick to reduce finite precision error (Num. Rec.)
c-----------------------------------------------------------------------
            dp = y(5) - ivs(2)
            y(6) = ivs(1) + dx
            dx = y(6) - ivs(1)
            y(7) = ivs(2)

c-----------------------------------------------------------------------
c     Define time interval and call lsoda
c-----------------------------------------------------------------------
            tbgn = 0.0d0
            tend = iN*T0
            istate = 1

            call lsoda(fcn,neq,y,tbgn,tend,itol,rtol,atol,itask,
     &           istate,iopt,rwork,lrw,iwork,liw,jac,jt)

c-----------------------------------------------------------------------
c     Error message if lsoda fails
c-----------------------------------------------------------------------
            if (istate .ge. -1) go to 100
            write(0,*) 'orbitnt: Error return ',istate,
     &           ' from integrator lsoda.'
            if (otrace) then
               close(ufrom)
            end if
            stop

 100        continue

c-----------------------------------------------------------------------
c     If lsoda is successful, calculate residual
c-----------------------------------------------------------------------
            res(1) = y(2) - ivs(1)
            res(2) = y(3) - ivs(2)
            nrm2res = dv12norm(res,npars)

c-----------------------------------------------------------------------
c     Define Jacobian for this Newton step
c-----------------------------------------------------------------------
            J(1,1) = (y(6)-y(2))/dx-1.0d0
            J(1,2) = (y(4)-y(2))/dp
            J(2,1) = (y(7)-y(3))/dx
            J(2,2) = (y(5)-y(3))/dp-1.0d0

c-----------------------------------------------------------------------
c     call dgesv to solve Jdx=res
c-----------------------------------------------------------------------
            call dgesv(npars,nrhs,J,npars,ipiv,res,npars,info)

c-----------------------------------------------------------------------
c     Error message if dgesv fails
c-----------------------------------------------------------------------
            if (info .ne. 0) then
               write(0,*) 'orbitnt: dgesv failed'
               if (otrace) then
                  close(ufrom)
               end if
               stop
            else

c-----------------------------------------------------------------------
c     if dgesv succeeds, calculate norms and update ivs
c-----------------------------------------------------------------------
               nrm2ivs = dv12norm(ivs,npars)
               nrm2upd = dv12norm(res,npars)

               do ipar = 1, npars
                  ivs(ipar) = ivs(ipar) - res(ipar)
               end do

c-----------------------------------------------------------------------
c     Tracing output to monitor convergence
c-----------------------------------------------------------------------
               write(0,200) iter, ivs(1), ivs(2),
     &              log10(max(nrm2upd,1.0d-60)),
     &              log10(max(nrm2res,1.0d-60))
 200           format(i2,1p,2e24.16,0p,2f8.2)

c-----------------------------------------------------------------------
c     Check for convergence
c-----------------------------------------------------------------------
               if (drelabs(nrm2upd,nrm2ivs,1.0d-6) .le. ntol)
     &              go to 300
            end if
         end do

c-----------------------------------------------------------------------
c     No-convergence exit
c-----------------------------------------------------------------------
         write(0,*) 'No convergence after ',mxiter, ' iterations'//
     &        ' for orbit N = ',iN
         if (otrace) then
            close(ufrom)
         end if
         stop

 300     continue

c-----------------------------------------------------------------------
c     Normal exit, set up values of y for final call to lsoda
c     Must call lsoda once more to calculate R and S
c-----------------------------------------------------------------------
         dx = max(EPS*abs(ivs(1)),1d-8)
         dp = max(EPS*abs(ivs(2)),1d-8)

         y(1) = 0.0d0
         y(2) = ivs(1)
         y(3) = ivs(2)
         y(4) = ivs(1)
         y(5) = ivs(2) + dp
         dp = y(5) - ivs(2)
         y(6) = ivs(1) + dx
         dx = y(6) - ivs(1)
         y(7) = ivs(2)

c-----------------------------------------------------------------------
```

```
c     If we are tracing the periodic orbit we will output the
c     coords. at several times
c-----------------------------------------------------------------------
         if (otrace) then
            do itout = 1, notpp*iN
               tout(itout) = 0.0d0 + (itout-1)*T0/notpp
            end do
            tout(notpp*iN) = iN*T0
            write(ufrom,*) ivs(1), ivs(2)
            do itout = 2, notpp*iN
               istate = 1
               tbgn = tout(itout-1)
               tend = tout(itout)
               call lsoda(fcn,neq,y,tbgn,tend,itol,rtol,atol,itask,
     &              istate,iopt,rwork,lrw,iwork,liw,jac,jt)
               if (istate .lt. -1) then
                  write(0,*) 'orbitnt: Error return ',istate,
     &                 ' from lsoda on last call for N = ',iN
                  close(ufrom)
                  stop
               else
                  write(ufrom,*) y(2), y(3)
               end if
            end do
            close(ufrom)
         else

c-----------------------------------------------------------------------
c     If we are not tracing the orbit, we can integrate in one step
c-----------------------------------------------------------------------
            istate = 1
            tbgn = 0.0d0
            tend = iN*T0
            call lsoda(fcn,neq,tbgn,tend,itol,rtol,atol,itask,istate,
     &           iopt,rwork,lrw,iwork,liw,jac,jt)
            if (istate .lt. -1) then
               write(0,*) 'orbitnt: Error return ',istate,
     &              ' from lsoda on last call for N = ',iN
               stop
            end if
         end if

c-----------------------------------------------------------------------
c     Compute R and S and write to standard out
c-----------------------------------------------------------------------
         R = (2.0d0 - (y(6)-y(2))/dx - (y(5)-y(3))/dp)/4.0d0
         S = y(1)
         write(*,*) iN, R, S
      end do

      stop

c-----------------------------------------------------------------------
c     usage message
c-----------------------------------------------------------------------
 900  continue

      write(0,*) 'usage: orbitnt <x0> <p0> [<ltol> <ntol>]'
      write(0,*)
      write(0,*) 'x0,p0: initial values in orbit search'
      write(0,*) 'ltol: optional tolerance for lsoda'
      write(0,*) 'ntol: optional tolerance for Newton convergence'

      stop

      end



c=======================================================================
c     dvl2norm: Returns l2-norm of double precision vector
c     (by Matt Choptuik)
c=======================================================================
      real*8 function dvl2norm(v,n)

         implicit none

         integer n
         real*8 v(n)
         integer i

         dvl2norm = 0.0d0
         do i = 1, n
            dvl2norm = dvl2norm + v(i)*v(i)
         end do
         if (n .gt. 0) then
            dvl2norm = sqrt(dvl2norm/n)
         end if

         return
      end
```

```
c=======================================================================
c     drelavs: Function useful for 'relativizing' quantity being
c     monitored for convergence (by Matt Choptuik)
c=======================================================================
      real*8 function drelabs(dx,x,xfloor)

         implicit none

         real*8 dx, x, xfloor

         if (abs(x) .lt. abs(xfloor)) then
            drelabs = abs(dx)
         else
            drelabs = abs(dx/x)
         end if

         return
      end

c=======================================================================
c     This is a subroutine that defined the equations of motion for
c     the driven harmonic oscillator system:
c
c     H = p**2/(2*m) + m*w0**2*x**2/2 + V*x*cos(w*t+theta)
c
c     The equations are set up to calculate the following quantities:
c
c     y(1) = S(x0,p0,t) (S is the action along the classical path)
c     y(2) = x(x0,p0,t)
c     y(3) = p(x0,p0,t)
c     y(4) = x(x0,p0+dp,t)
c     y(5) = p(x0,p0+dp,t)
c     y(6) = x(x0+dx,p0,t)
c     y(7) = p(x0+dx,p0,t)
c=======================================================================
      subroutine fcn(neq, t, y, yprime)

         implicit none

c-----------------------------------------------------------------------
c     Common block for communicating with orbitnt, jac
c-----------------------------------------------------------------------
         include 'fcn.inc'

c-----------------------------------------------------------------------
c     neq: number of odes (7)
c     t: time variable
c     y: solution of odes
c     yprime: derivative of solution
c-----------------------------------------------------------------------
         integer neq
         real*8 t
         real*8 y, yprime
         dimension y(1), yprime(1)

c-----------------------------------------------------------------------
c     Define derivatives
c-----------------------------------------------------------------------
         yprime(1) = y(3)**2/(2.0d0*m) - m*w0**2*y(2)**2 -
     &        V*y(2)*cos(w*t+theta)
         yprime(2) = y(3)/m
         yprime(3) = -m*w0**2*y(2) - V*cos(w*t+theta)
         yprime(4) = y(5)/m
         yprime(5) = -m*w0**2*y(4) - V*cos(w*t+theta)
         yprime(6) = y(7)/m
         yprime(7) = -m*w0**2*y(6) - V*cos(w*t+theta)

         return
      end


c=======================================================================
c     Optional Jacobian for lsoda
c
c     Note: in 'orbitnt.f' lsoda is set up for a banded Jacobian with
c     ml=1 and mu=2
c=======================================================================
      subroutine jac(neq,t,y,ml,mu,pd,nrowpd)

         implicit none

c-----------------------------------------------------------------------
c     Common communication with orbitnt and fcn
c-----------------------------------------------------------------------
         include 'fcn.inc'

c-----------------------------------------------------------------------
c     neq: number of odes (7)
c     ml: number of lower diagonals in Jacobian (1)
c     mu: number of upper diagonals in Jacobian (2)
c     nrowpd: number of rows in array pd (nrowpd=ml+mu+1=4)
c-----------------------------------------------------------------------
```

```
         integer neq
         integer ml, mu, nrowpd

c-----------------------------------------------------------------------
c      t: time variable
c      y: solution of odes
c      pd: array of partial derivatives (Jacobian)
c-----------------------------------------------------------------------
         real*8 t
         real*8 y, pd
         dimension y(1), pd(nrowpd,1)

c-----------------------------------------------------------------------
c      Define partial derivatives
c
c      note J(i,j) = pd(i-j+mu+1,j)  (banded matrix form)
c-----------------------------------------------------------------------
         pd(2,2) = -2.0d0*m*w0**2*y(2) - V*cos(w*t+theta)
         pd(1,3) = y(3)/m
         pd(2,3) = 1.0d0/m
         pd(4,2) = -m*w0**2
         pd(2,5) = 1.0d0/m
         pd(4,4) = -m*w0**2
         pd(2,7) = 1.0d0/m
         pd(4,6) = -m*w0**2

         return
      end

c=======================================================================
c Common block for use with 'orbitnt'.  Shares values of
c parameters among 'orbitnt', 'fcn', and 'jac'.  The parameters
c are:
c
c w: frequency of driving field
c theta: initial phase of driving field
c V: strength of driving field
c m: mass of particle
c W: harmonic oscillator frequency
c=======================================================================
      real*8
     & w,theta,  m,
     & V,w0

      common /com_fcn/
     & w,theta,m,
     & V,w0
```

7.6 Plotting Quasienergy Density of States

```
c=======================================================================
c      dhonqp.f: This routine generates a plot of the quasienergy
c      density of states for the driven harmonic oscillator, using
c      results from the program 'orbitnt.f'.
c=======================================================================
      program dhonqp

      implicit none

c-----------------------------------------------------------------------
c      w: driving frequency
c      T: driving period
c      Q: quasienergy value
c      R(j): residue of the jth periodic orbit
c      S(j): action "
c      imax: total number of output points
c      jmax: number of orbits used in calculating density of states
c-----------------------------------------------------------------------
      real w, pi, T, Q, hbar, dQ, Rt, St, b

      integer i, imax, j, jmax, N
      parameter (imax = 5000, jmax = 60)

      real R(jmax), S(jmax)

      pi = 4.0d0*atan(1.0d0)
      w = 4.0d0*pi
      T = 2.0d0*pi/w
      hbar = 1.0d0

c-----------------------------------------------------------------------
c      read in output from 'orbitnt.f'
c-----------------------------------------------------------------------
      do j = 1, jmax
         read(*,*) N, Rt, St
         if (N .eq. j) then
            R(j) = Rt
            S(j) = St
         else
            write(0,*) 'Error reading input'
         end if
      end do

c-----------------------------------------------------------------------
c      calculate the density of states for each value of Q and output
c      to standard out
c-----------------------------------------------------------------------
      do i = 1, imax
         Q = 0.0d0 + dble(i-1)*w*hbar/dble(imax-1)
         b = 0.0d0
         do j = 1, jmax
            b = b - T*cos((dble(j)*Q*T+S(j))/hbar)/(2.0d0*pi*
     &           hbar*sqrt(R(j)))
         end do
         write(*,*) Q, b
      end do

      stop

      end

c=======================================================================
c      dhoqp.f: This routine generates a plot of the quasienergy
c      density of states for the driven harmonic oscillator using
c      the results of analytical calculation.
c=======================================================================
      program dhoqp

      implicit none

c-----------------------------------------------------------------------
c      w: driving frequency
c      w0: natural frequency of the oscillator
c      E: quasienergy value
c      V: strength of the driving field
c      imax: number of output points
c      jmax: number of periodic orbits used in calculating the density
c            of states
c-----------------------------------------------------------------------
      real w, w0, a, b, E, dE, V, pi, m, hbar

      integer i, imax, j, jmax
      parameter(imax = 5000, jmax = 60)

      pi = 4.0d0*atan(1.0d0)
      w = 4.0d0*pi
      w0 = 4.0d0
      V = 1.0d-1
      m = 1.0d0
      hbar = 1.0d0

c-----------------------------------------------------------------------
c      calculate the density of states for each E and write to
c      standard out
c-----------------------------------------------------------------------
      do i = 1, imax
         E = 0.0d0 + dble(i-1)*w*hbar/dble(imax-1)
         b = 0.0d0
         do j = 1, jmax
            a = 2.0d0*pi*j*(E-V**2/(4.0d0*m*(w**2-w0**2)))/
     &           (w*hbar)
            b = b - 2.0d0*cos(a)/(w*hbar*sqrt(2.0d0*(1.0d0
     &           - cos(2.0d0*pi*j*w0/w))))
         end do
         write(*,*) E, b
      end do

      stop

      end

c=======================================================================
c      diffp.f: This routine generates a graph of the difference
c      between the outputs of 'dhonqp.f' and 'dhoqp.f'.
c=======================================================================
      program dhonqp

      implicit none

c-----------------------------------------------------------------------
c      w: driving frequency
c      T: driving period
c      Q: quasienergy value
c      R(j): residue of jth periodic orbit
c      S(j): action "
c      w0: natural frequency of the oscillator
c      V: strength of driving field
c      imax: number of output points
c      jmax: number of orbits used in calculating the density of states
c-----------------------------------------------------------------------
      real w, pi, T, Q, hbar, dQ, Rt, St, b
```

```
      real c, w0, a, V, m

      integer i, imax, j, jmax, N
      parameter (imax = 5000, jmax = 60)

      real R(jmax), S(jmax)

      pi = 4.0d0*atan(1.0d0)
      w = 4.0d0*pi
      T = 2.0d0*pi/w
      hbar = 1.0d0
      w0 = 4.0d0
      V = 1.0d-1
      m = 1.0d0

c-----------------------------------------------------------------------
c     read in output from 'orbitnt.f'
c-----------------------------------------------------------------------
      do j = 1, jmax
         read(*,*) N, Rt, St
         if (N .eq. j) then
            R(j) = Rt
            S(j) = St
         else
            write(0,*) 'Error reading input'
         end if
      end do

c-----------------------------------------------------------------------
c     calculate density of states with the two methods and write the
c     difference to standard out
c-----------------------------------------------------------------------
      do i = 1, imax
         Q = 0.0d0 + dble(i-1)*w*hbar/dble(imax-1)
         b = 0.0d0
         c = 0.0d0
         do j = 1, jmax
            b = b - T*cos((dble(j)*Q*T+S(j))/hbar)/(2.0d0*pi*
     &           hbar*sqrt(R(j)))
            a = 2.0d0*pi*j*(Q-V**2/(4.0d0*m*(w**2-w0**2)))/
     &           (w*hbar)
            c = c - 2.0d0*cos(a)/(w*hbar*sqrt(2.0d0*(1.0d0
     &           - cos(2.0d0*pi*j*w0/w))))
         end do
         write(*,*) Q, b-c
      end do

      stop

      end
```

7.7 Strobe Plots of Driven Triangle Well

```
c=======================================================================
c     twsplt:  This program generates a strobe plot of the driven
c     triangle well system.  This system has a Hamiltonian
c     H = p**2/(2*m) - V0*x - V*x*cos(w*t+theta)     x < 600
c     The system is strobed at the period of the driving field.
c     Because of the presence of the infinite wall the program
c     do2cje from the NAG library is used.  This program integrates
c     the equations of motion until the particle hits the wall.
c     The momentum is then reversed and the equations of motion
c     are again integrated until another collision, etc.
c=======================================================================
      program twsplt

      implicit none

c-----------------------------------------------------------------------
c     n: number of odes
c     iwork: size of work array used by d02ejf
c-----------------------------------------------------------------------
      integer n, iwork
      parameter (n = 2, iwork = 28 + 21*n)

c-----------------------------------------------------------------------
c     tbgn: initial time of integration (overwritten by do2ejf)
c     tend: final time
c     tol: tolerance for d02cje
c-----------------------------------------------------------------------
      real   tbgn, tend, tdt, dt
      parameter (dt = 1.0e-4)
      real   tol, default_tol
      parameter (default_tol = 1.0e-8)

c-----------------------------------------------------------------------
c     i: index used to increment output times
c     ifail: error return from do2cje
c-----------------------------------------------------------------------
      integer ifail

c-----------------------------------------------------------------------
c     work: work array used by d02ejf
c     y: solution vector
c     g: function that monitors collisions with the wall
c-----------------------------------------------------------------------
      real   work(iwork), y(n)
      real   g

c-----------------------------------------------------------------------
c     pmax: maximum value of p used as initial condition
c     xmin: minimum value of x "
c     np: number of initial values of p
c     nx: "                             x
c     niter: number of strobe times (approx)
c-----------------------------------------------------------------------
      real pmax, xmin
      integer np, nx, niter
      integer ip, ix

c-----------------------------------------------------------------------
c     argument parsing variables
c-----------------------------------------------------------------------
      real r8arg, r8_never
      parameter (r8_never = -1.0e-60)
      integer iargc

c-----------------------------------------------------------------------
c     fcn: subroutine that calculates deriviatives
c     pederv: "   "     "  Jacobian
c     out: subroutine that outputs solution at specified times
c-----------------------------------------------------------------------
      external g, d02cje, fcn, out, cjwd02, cjxd02

c-----------------------------------------------------------------------
c     Common communication with subroutines
c-----------------------------------------------------------------------
      include 'fcn.inc'

      real   pi

c-----------------------------------------------------------------------
c     parse command line arguments
c-----------------------------------------------------------------------
      if (iargc() .lt. 2) go to 900

      w = r8arg(1, r8_never)
      v = r8arg(2, r8_never)
      if (w .eq. r8_never .or. v .eq. r8_never) go to 900

      tol = r8arg(3, default_tol)
      if (tol .le. 0.0e0) tol = default_tol

c-----------------------------------------------------------------------
c     Set values of parameters
c-----------------------------------------------------------------------
      v0 = 4.0e-1
      m = 1.0e0
      theta = 0.0e0
      niter = 200
      pi = 4.0e0*atan(1.0e0)
      per = 2.0e0*pi/w
      tend = 2.0e0*niter*per
      l = 6.0e2

c-----------------------------------------------------------------------
c     set iteration parameters
c-----------------------------------------------------------------------
      pmax = 2.1e1
      xmin = 5.4e2
      np = 1
      nx = 40

c-----------------------------------------------------------------------
c     iterate over initial conditions
c-----------------------------------------------------------------------
      do ip = 1, np

         do ix = 1, nx

c-----------------------------------------------------------------------
c     Initialize solution at t = 0
c-----------------------------------------------------------------------
            tbgn = 0.0e0
            y(1) = xmin + real(ix-2)*(l-xmin)/real(nx-1)
            y(2) = 0.0e0

            i = 1

 100        continue
```

```
c-----------------------------------------------------------------------
c     Call d02ejf to integrate y until collsion with wall
c-----------------------------------------------------------------------
            ifail = 0

            call d02cje(tbgn,tend,n,y,fcn,tol,'D',out,g,work,ifail)

c-----------------------------------------------------------------------
c     Error message if d02cje fails
c-----------------------------------------------------------------------
            if (ifail .gt. 0) then
               write(0,*) 'd02cje returns ifail = ',ifail
               stop
            end if

c-----------------------------------------------------------------------
c     Reverse momentum and return to d02cje unless at the end
c-----------------------------------------------------------------------
            if (tbgn .lt. niter*per) then
               y(2) = -y(2)
               ifail = 0
               i = i - 1
               tdt = tbgn + dt
               call d02cje(tbgn,tdt,n,y,fcn,tol,'D',cjxd02,cjwd02,
     &              work,ifail)
               go to 100
            end if
         end do
      end do

      stop

c-----------------------------------------------------------------------
c     usage message
c-----------------------------------------------------------------------
 900  continue

      write(0,*) 'usage: twsplt <w> <V> [<tol>]'
      write(0,*)
      write(0,*) 'w: frequency of driving field'
      write(0,*) 'V: strength of driving field'
      write(0,*) 'tol: optional tolerance for d02cje'

      stop

      end

c=======================================================================
c     this is a function that monitors collisions with the infinite
c     wall in the driven triangle well
c=======================================================================
      real function g(t, y)

      implicit none

c-----------------------------------------------------------------------
c     common commmunication with main program
c-----------------------------------------------------------------------
      include 'fcn.inc'

      integer n
      parameter (n = 2)
      real t
      real y(n)

      g = y(1) - l

      return

      end

c=======================================================================
c     This routine defines the equations of motion for the driven
c     triangle well.  It is used with twsplt.f
c=======================================================================
      subroutine fcn(t, y, f)

      implicit none

c-----------------------------------------------------------------------
c     common communication with main program
c-----------------------------------------------------------------------
      include 'fcn.inc'

c-----------------------------------------------------------------------
c     n: number of odes
c     t: time independent variable
c     y: solution (y(1)=x, y(2)=p)
c     f: derivative of solution
c-----------------------------------------------------------------------
      integer n
      parameter (n = 2)
      real t
      real y(n), f(n)

c-----------------------------------------------------------------------
c     evaluate derivatives
c-----------------------------------------------------------------------
      f(1) = y(2)/m
      f(2) = v0 + v*cos(w*t + theta)

      return

      end

c=====================================================================
c This is a common block include file for use with the routine
c twsplt.f.  The parameters are:
c
c i: index for output times
c V: strength of driving field
c V0: strength of triangle well
c w: frequency of driving field
c theta: phase "
c Per: period "
c m: mass of particle
c L: position of infinite wall
c=====================================================================
      integer i
      real*8 V, V0, w, theta, L, Per, m

      common /com_fcn/
     & i,
     & V,V0,w,theta,
     & L,Per,m
```

References

[1] M. Ferray, A. L'Huillier, X.F. Li, L.A. Lompre, G. Mainfray, and C. Manus. *Journal of Physics*, B(21):L31, 1988.

[2] S. Cocke and L.E. Reichl. High-harmonic generation in a driven triangular well: The implications of chaos. *Physical Review*, A(53):1746, 1996.

[3] K.C. Kulander, K.J. Schafer, and J.L. Krause. Dynamics of short-pulse excitation, ionization and harmonic conversion. In *Super-Intense Laser-Atom Physics*, volume 316 of *Series B: Physics*, page 95, New York, 1993. NATO Advanced Science Institute, Plenum Press.

[4] M.C. Gutzwiller. *Chaos in Classical and Quantum Mechanics*. Springer-Verlag, New York, 1990.

[5] M. Tabor. A semiclassical quantization of area-preserving maps. *Physica*, D(6):195, 1983.

[6] F. Benvenuto, G. Casati, I. Guarneri, and D.L. Shepelyansky. A quantum transition from localized to extended states in a classically chaotic system. *Zeitschrift für Physik*, B(84):159, 1991.

[7] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in FORTRAN 77*. Cambridge University Press, New York, second edition, 1992.

[8] R. Balian and C. Bloch. Solution of the schrödinger equation in terms of classical paths. *Annals of Physics*, page 514, 1974.