Large Mass Approximations in Quantum Physics and A Bridge to Quantum Chemistry

Davin M. Potts

University Undergraduate Research Fellow, 1994-95

Texas A&M University

Departments of Mathematics and Chemistry

APPROVED

Undergraduate Advisor Stephen A. Fulling

Undergraduate Advisor Danny L Yeager

Exec. Dir., Honors Program

Table of Contents

I. Introduction to Propagators

A. The Large Mass Asymptotic Propagator

The simplest description of propagators is this: propagators are tools for solving differential equations. Solutions to the differential equations we will be discussing are functions. If one knows a solution at a particular point, time or state, a propagator can "propagate" this particular value of the solution to other times or states. That is, from a known value of a solution function, one can reconstruct the entire solution function. This is how propagators get their name.

The fact that the entire solution can be reconstructed by the propagator proves that all the "information" contained within the solution is also contained within the propagator. Depending upon the differential equation, the propagator will contain additional and potentially useful information which is not contained in the solution. This additional information provides the driving force behind calculating propagators for the partial differential equations we will be working with.

The time-dependent Schrödinger equation,

$$-\frac{\hbar}{i}\frac{d}{dt}\Psi(x,t) = \hat{H}\,\Psi(x,t)$$

is a partial differential equation which describes some physical system, depending upon the definition of $\hat{H}$. The wavefunction Ψ is the unknown in this equation, x and t are variables representing position and time, and the remaining terms are constants. Ψ is a

function of position and time which describes the state of the physical system defined by $\hat{H}$. By expanding $\hat{H}$ in the time-dependent Schrödinger equation, we get the standard form of the equation,

$$-\frac{\hbar}{i}\frac{\partial\,\Psi(x,t)}{\partial\,t} = -\frac{\hbar^2}{2m}\nabla^2\,\Psi(x,t) + \lambda\,V(x,t)\,\Psi(x,t)$$
$$\equiv \hat{H}\,\Psi(x,t)$$

where ∇^2 is the Laplacian, $V(x,t)$ is the potential function, m is mass and λ is the coupling constant.

The propagator $K(x, y; t; \hbar, m, \lambda)$ is defined to be the integral kernel of the operator $e^{-it\hat{H}/\hbar}$. Computing this integral kernel is far from being straight-forward. Therefore, if we rewrite the last equation in the following manner,

$$\frac{t\hat{H}}{\hbar} = -\frac{\hbar\,t}{2\mathrm{m}}\nabla^2 + \frac{\lambda\,t}{\hbar}V$$
$$\equiv -\tfrac{1}{2}\,A\,\nabla^2 + B\,V$$

we see that there are two parameters with which to form a perturbative expansion for the propagator. One can expand the propagator in any of the following ways:

$A\to 0$	$(m\to\infty)$	a semiclassical expansion (an expansion in derivatives of V)
$B\to 0$	$(\lambda\to 0)$	the coupling-constant expansion (an expansion in powers of V)
$A, B\to 0$	$(t\to 0)$	the small-time expansion
$\frac{A}{B}\to 0$	$(\hbar\to 0)$	a semiclassical expansion (leads to the well-known WKB expansion)[1]

In the following work, we will use the large mass expansion, A→0. The different expansions each have advantages and disadvantages. Formulas for computing the large mass expansion have been derived by others[2], but never tested. The primary purpose of this work is to examine the usefulness, the advantages and disadvantages of the large mass expansion.

B. The Energy-Domain Green's Function

Related to the time-dependent Schrödinger equation via a Fourier transform is the time-independent Schrödinger equation,

$$\hat{H}\,\psi(x) = E\,\psi(x)$$

Again, $\hat{H}$ describes some physical system and ψ is an unknown. The major differences are that now, both $\hat{H}$ and ψ are independent of time (t) and another term, E, appears. E is also an unknown. It contains the eigenvalues of this partial differential equation. These eigenvalues correspond to the quantized energy levels of the physical system defined by $\hat{H}$. Rather than finding a time-dependent propagator, typically the energy-domain Green's function is used. The energy-domain Green's function is the propagator for this differential equation.

The primary reason for transforming the time-dependent into the time-independent Schrödinger equation is to find the quantized energy levels of the physical system, i.e. to find E. Thus, in certain situations, finding E is more important than solving for ψ. For

the Green's function we will describe, E is part of the additional information contained within the propagator.

II. Computational Implementation of the Large Mass Expansion

A. Electromagnetic Field Expansion / C code

We will first use the large mass gauge invariant asymptotic expansion of the propagator given by Osborn et al.[2]:

$$K(x, y; t; \hbar, m, \lambda) = \left(\frac{m}{2\pi i\hbar t}\right)^{d_2} Exp\left[\frac{im(x-y)^2}{2\hbar t}\right] * \left\{Exp\left[\tfrac{1}{i\hbar} J(x, y; t)\right]\right\}\left\{1 + \tfrac{1}{m} T_1(x, y; t) + \tfrac{1}{m^2} T_2(x, y; t) + \cdots\right\}$$

This is the propagator for the Hamiltonian:

$$\hat{H} = \frac{1}{2m}\left(\frac{\hbar}{i}\nabla - \frac{e}{c}a\right)^2 + ev$$

Osborn provides formulas for computing this expansion up to second order (formulas for J, T_1 and T_2). This particular expansion allows for any vector potential; that is, an electromagnetic potential. One can slightly improve upon the above formula for the propagator by rewriting it as:

$$K(x, y; t; \hbar, m, \lambda) = \left(\frac{m}{2\pi i\hbar t}\right)^{d_2} Exp\left[\frac{im(x-y)^2}{2\hbar t}\right] *$$

$$Exp\left[\frac{1}{i\hbar} J(x, y; t) + \frac{1}{m} J_1(x, y; t) + \frac{1}{m^2} J_2(x, y; t) + \cdots\right]$$

where $J_1 = T_1$ and $J_2 = T_2 - \frac{1}{2}T_1^2$. Complete formulas are contained in Appendix A.

1. Development and Organization

We programmed the formulas for computing $K(x, y; t; \hbar, m, \lambda)$ in the computer language C. As a result of Osborn's formulation, T_2 and T_1 (or J_2 and J_1) require as input vectors x and y, scalar t, the scalar potential v and the vector potential a. From these inputs we derive a slew of terms, essentially different derivatives of the functions: f, matrixA and Omega. Organization of the C source code files is as follows:

flat.h – Include file containing prototypes for the functions used in calculating T_1, T_2, J_1, J_2 and J. Also contains definitions for all structures used.

numintg.c – Contains functions which employ Simpson or Romberg integration schemes on arbitrary functions. Through the use of pointers, these functions are not specific to any one function, thus allowing them to be reused to integrate different functions. Contains functions to carry out single, double or triple integrals.

qsederiv.c – Contains explicit definitions for the derived terms. If explicit definitions are not available, qsenumer.c can be used instead to numerically derive them. *qsederiv.c* is preferred because it is faster and more accurate.

qsefuncs.c – Contains definitions for the Green's function g, the linear path w, Omega, a max function and d1g.

not necessary to utilize these C subroutines. Simply understand that these functions are returning more than just a single value.

2. Instructions for Future Use

For a specific choice of a, v, y, x and t, the following must be done by the user:

1) Modify *flat.h* so that the number of dimensions being considered is accurate. Note that these routines are general for any number of dimensions desired.

2) Modify *qseinput.c* so that a and v are defined appropriately.

3) If functions defining f, MatrixA, divf, GradOmega, GradA, LaplacianOmega, LapdivOmega, GraddivOmega are known, modify *qsederiv.c* appropriately.

4) In the user's code, call the function(s) T2, T1, J2, J1, J in a manner similar to that used in the sample C file *tjmain.c* (pages B-54 - B-56). Or look at the file *flat.h* to see its prototype.

5) Compile and link the program with either *qsederiv.c* or *qsenumer.c* (not both) depending upon whether the derived functions are known explicitly or not.

B. Connected Graph Expansion

If we ignore the possibility of having a time varying or a vector potential, higher order terms in the large mass asymptotic expansion are much easier to calculate. This is done using a general graph expansion of the propagator[3,4,5]:

$$K(x, y; t; \hbar, m) = \left(\frac{2\pi i \hbar t}{m}\right)^{-\frac{d}{2}} e^{im|x-y|^2/2\hbar t} e^{L}$$

where L has the asymptotic expansion:

$$L \sim \sum_{p=1}^{\infty} \sum_{q=0}^{\infty} (-i)^{p+q} A^q B^p \sum_{G \in \mathbf{G}_{pq}^C} L_{pq}[G]$$

$$L_{pq}[G] \equiv \left[2^n p! \prod_{\alpha=1}^{\frac{1}{2}p(p+1)} l_\alpha ! \right]^{-1} \int_{[0,1]^p} d^p \xi \prod_{\alpha} \left[g(\xi_{i_\alpha}, \xi_{j_\alpha}) \nabla_{i_\alpha} \nabla_{j_\alpha} \right]^{l_\alpha} \prod_{i=1}^{p} V(z_i)$$

where

$$z_i \equiv y + \xi_i (x - y)$$
$$g(\xi_{i_\alpha}, \xi_{j_\alpha}) \equiv \xi_< (\xi_> - 1)$$

1. Development and Organization

This expansion is represented as a sum over connected graphs. In our computations, we use the conventions of Fulling and Borosh[3]. Specifically, we use their adjacency matrix representation for graphs and their Mathematica programs for generating adjacency matrices. We have written Mathematica routines to symbolically calculate the propagator, using the above formula. Of particular interest for separate application is our Mathematica routine, *ConnGraphQ*, which tests an adjacency matrix for connectivity.

By computing the propagator symbolically instead of numerically, as the C code does, we gain versatility. A single calculation of a symbolic formula for the propagator may be used for further symbolic computation, many different graphical plots, etc. A single numerical calculation with the C code finds use in a single graphical plot. However, the formula for the propagator used in the C code is not easily programmed into or calculated by Mathematica. Fortunately, the connected graph expansion is

2. Instructions for Future Use

A complete listing of the Mathematica routines, with documentation, is contained in Appendix C. A basic understanding of the Mathematica environment is required to productively use and manipulate the output from these routines. The two routines of greatest importance are *L* and *LExpSeries*. *L* calculates individual terms in the expansion. *LExpSeries* calculates the series for a given set of {p,q} values. Note that both routines are limited and can only work in 1-dimensional coordinate space.

LExpSeries takes advantage of a major computational simplification: by moving the summation over connected graphs (a finite sum for a given p and q) inside the integration over p-dimensions from 0 to 1, we reduce the integration to that over a single simplex (see A-3 - A-5 for a brief discussion) and multiply by the constant (p!). The user is also encouraged to experiment with *ConnGraphQ*.

III. Computational Results for the Large Mass Expansion

Our routines were tested on a series of cases: linear potential, constant magnetic potential, sinusoidal potential, harmonic oscillator potential and inverse quadratic potential. Only the first case and last two cases have purely scalar potentials and may therefore be calculated using the Mathematica routines. All cases were examined using the C code. Results from our routines were analyzed graphically. We developed a systematic and informative scheme of plotting data from our calculations: for a fixed time

t and initial position y, let final position x vary from -5 to 5. For the potentials we examined, Appendix D contains many such plots.

A. Linear Potential

The linear potential is an exactly solvable case[6]. For example, $v[x,t] = 2x_1 - 1$ with $a[x,t] = \{0,0,0\}$. Here, $\frac{1}{i\hbar} J + \frac{1}{m} J_1 + \frac{1}{m^2} J_2$ forms a straight line. While not a system of particular physical interest, it does provide a means of testing the programs we have written for accuracy. Because of the trivial nature of this potential, no plots are included.

B. Constant Magnetic Potential

Based upon preliminary results, we did not concentrate on the constant magnetic potential. For example, $v[x,t] = 0$ with $a[x,t] = \{-\frac{1}{2} x_2, \frac{1}{2} x_1, 0\}$. The coefficients in our truncated expansion are so small for the cases we tested that the difference between the approximation and the free propagator is hardly noticeable. A brief graphical examination of the constant magnetic potential may be found on pages D-3 - D-10.

C. Sinusoidal Potential

The sinusoidal potential is an electromagnetic potential. The exact form of the propagator is not known for this case. This is the only time-dependent potential we studied. We examine $v[x,t] = 0$ with $a[x,t] = \{0,\ 2Sin[x_1 - t],\ 0\}$. The plots of the data from the C code indicates that our truncated approximation is not a large enough expansion to accurately approximate the propagator. Increasing from first to

second order greatly affects the approximation for our test cases. The plots contained on pages D-11 - D-22 demonstrate this change.

D. Harmonic Oscillator (Quadratic) Potential

For the harmonic oscillator potential, also known as the quadratic potential, the exact propagator is known. Thus we can compare our computed results to the true propagator. We examine $v[x, t] = \frac{1}{2} x_1^2$ and $a[x, t] = \{0,0,0\}$. The true propagator was given by Farina et al.[7] to be:

$$K(x, y; t; \hbar, m, \omega) = \sqrt{\frac{m\omega}{2\pi i \hbar Sin[\omega t]}} * Exp\left[\frac{im\omega}{2\hbar Sin[\omega t]}\left[(x^2 + y^2)Cos[\omega t] - 2xy\right]\right]$$

$$\text{where } \omega = \sqrt{\frac{2}{m}}$$

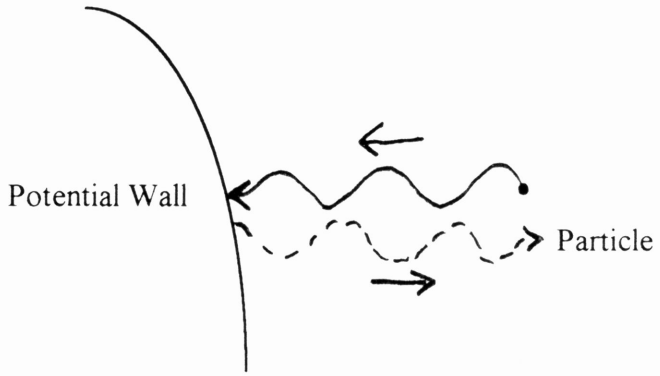
See pages D-23 - D-32 for an extensive graphical analysis of this case. Each plot compares the first and second order approximations with the true propagator.

E. Inverse Quadratic Potential

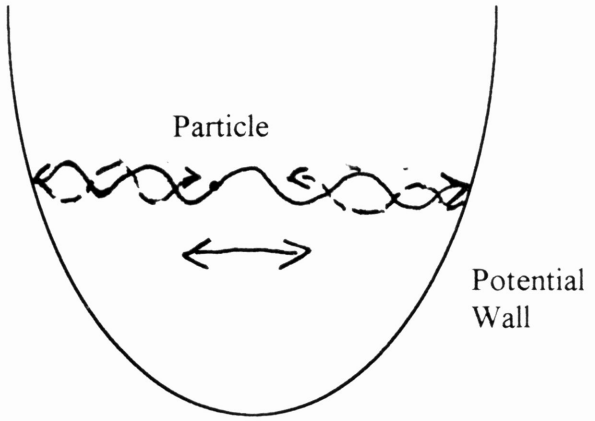
We examine $v[x, t] = -\frac{1}{2} x_1^2$ and $a[x, t] = \{0,0,0\}$. With a change in sign, Farina's formula for the harmonic oscillator provides the exact propagator for this case as well. A comparison with the harmonic oscillator potential demonstrates interesting properties of the asymptotic expansion we are using. Pages D-32 - D42 contain many graphical comparisons between the two cases.

F. A Comparison of the Quadratic Potentials

The large mass expansion is inherently imperfect. The integrals in the formulas for the propagator have been simplified to integrals over straight lines. This is an approximation to the exact integrals over the exact classical trajectories (the $\hbar \rightarrow 0$ expansion)[1]. The major side effect that results is the propagator's inability to handle reflections. That is, imagine a particle being propagated towards a wall of the potential and the reflection that results:



So for the inverse quadratic potential, the particle is "outside" the potential walls and only one reflection results from "hitting" the wall. For the harmonic oscillator potential, the particle is "inside" the potential walls and many reflections result from hitting the wall:



This difference suggests that our semiclassical expansion should be better for the inverse quadratic case than for the harmonic oscillator case. This is confirmed by the series of plots contained in Appendix D on pages D-32 - D-42.

Another interesting behavior of the large mass expansion is visible on these plots. For the harmonic oscillator, all classical trajectories leaving the same initial point (at the same initial time) refocus at a single point at the same time. The classical trajectories refocus every integral multiple of half the period ($\frac{1}{2}(2\pi) = \pi$). Thus, in the progression of plots we see that the approximations and the true propagator become increasingly oscillatory as time is increased. When time is increased past t=3.1415..., that is, on the t=4 plot, the true propagator has suddenly "calmed" down. Its amplitude and period of oscillation suddenly calms down yet the first and second order approximations do not. The straight line trajectories used in the large mass expansion will never reconverge. Therefore the large mass expansion can not properly model this behavior of the true propagator.

IV. Bridges to Quantum Chemistry

Developing and running quantum chemistry computer programs is a growing industry. Many programs calculate electronic properties of chemical systems from the first principles of physics and bound state quantum mechanics, commonly referred to as *ab initio* calculations. The theory behind these computer algorithms is largely based upon the time-independent Schrödinger equation. Of the many methods employed, the use of

energy-domain Green's functions is most related to our time-dependent propagator. We have devised two possible schemes for extracting electronic properties from our large mass asymptotic expansion.

A. Quantum Chemistry Computational Conventions

The quantum chemistry codes that employ Green's functions typically use a finite set of basis functions to describe all of physical space. This is akin to truncating an infinite basis set for an infinite-dimensional vector space. In the codes, eigenvalue problems are set up and solved, often by self-consistent methods of iteration. While much of the theory used in the energy-domain Green's function codes originates from the time-dependent propagator, only energy dependent terms are used in developing the computer algorithms.

The poles of the energy-domain Green's function correspond to the excitation energy levels of a chemical system. Oddershede, Jørgensen and Yeager derived the following formula for an energy-domain Green's function[8]:

$$G(E) \equiv \left\langle\left\langle P; Q \right\rangle\right\rangle^{r}_{E+i\varepsilon} = \lim_{\varepsilon \to 0^+} \sum_n \left\{ \begin{array}{l} \dfrac{\langle 0|P|n\rangle\langle n|Q|0\rangle}{E - E_n + E_0 + i\varepsilon} \\ \\ -\dfrac{\langle 0|Q|n\rangle\langle n|P|0\rangle}{E + E_n - E_0 + i\varepsilon} \end{array} \right\}$$

Note that when $E = \pm(E_n - E_0)$, the Green's function has a pole. E_n is the n^{th} excited state energy and E_0 is the ground state energy. We will not be using this formulation of the Green's function, but we will attempt at a basic level to mimic its computational implementation.

B. Implementation of the Large Mass Expansion

We devised two different methods for extracting excitation energies from our time-dependent large mass asymptotic expansion of the propagator:

Method 1: Imitate the Hartree-Fock Green's Function method.

1) Select a finite set of basis functions, $\{\phi_n\}$.

2) Use a Fast Fourier Transform or an approximate method to numerically calculate $K(x, y; E)$ from $K(x, y; t)$.

3) Calculate the matrix $\langle \phi_i | K(x, y; E) | \phi_j \rangle$.

4) Self-consistently solve for the vector Φ : $\left(\langle \phi_i | K(x, y; E) | \phi_j \rangle \right)^{-1} \Phi = 0$. The poles will correspond to the zeroes in the inverted matrix.

Method 2: Search for the poles directly.

Using a Newton-Raphson technique, search either

1) $\mathrm{FFT}\big(K(x, y; t)\big) = K(x, y; E)$ for poles or

2) $K(x, y; t)$ for $\mathrm{FFT}^{-1}(\mathrm{pole})$.

C. Discussion of the Framework

While our idea of using the large mass expansion for electronic properties calculations is original, the idea of applying other semiclassical expansions is not. Delos and coworkers have done significant work on using semiclassical expansions to calculate

the excitation energies of chemical systems[9,10,11]. Makri and Miller use the short time semiclassical expansion for scattering theory calculations[12,13].

While Delos is able to successfully apply semiclassical expansions to the hydrogen atom, the expansions do not do well for larger chemical systems, i.e. helium and larger atoms or molecules. Miller uses the short time expansion to approximate some scattering terms which are otherwise too difficult to calculate using conventional scattering theory methods. The past experience of others and the inherent weaknesses of the large mass expansion strongly suggests that the methods outlined above will not accurately determine electronic properties for chemical systems.

All hope is not lost however. The idea of mimicking Miller's strategy for applying semiclassical methods holds some promise. Values or concepts too difficult to calculate using conventional bound state quantum mechanical methods could be approximated using the large mass expansion. Now that the background has been developed, determining where the large mass expansion could be employed is the next step. This is worthy of future effort.

V. Conclusions

After viewing the computational results from the large mass expansion, we see that our truncated expansion needs to be extended. Higher order terms in the expansion need to be calculated in order to further study the effectiveness and the properties of the large mass expansion. The newly written Mathematica code provides the means for calculating

higher order terms, with the constraints that we can no longer work with time-dependent or vector potentials and that we must work in 1-dimensional space.

In order to better model the behavior of the true propagators, it should be worthwhile to try using Padé Approximants. This means rewriting our power series expansion as a ratio of power series expansions:

$$\left(c_0 + c_1 \frac{1}{m} + c_2 \frac{1}{m^2} + c_3 \frac{1}{m^3} + \cdots\right) = \frac{\left(a_0 + a_1 \frac{1}{m} + a_2 \frac{1}{m^2} + \cdots\right)}{\left(1 + b_1 \frac{1}{m} + b_2 \frac{1}{m^2} + \cdots\right)}$$

We now recognize that the very nature of the large mass expansion severely limits its application to electronic properties calculations. However, now that we have a basic understanding of these limitations, the possible application of the large mass expansion to approximating computationally difficult values in quantum chemistry methods deserves further investigation.

Overall, we conclude that the programs and tools we have developed to this point are performing correctly and do give useful results for studying the large mass expansion.

[1] T. A. Osborn, F. H. Molzahn, *Phys. Rev. A.* **34**, 1696 (1986).

[2] L. Papiez, T. A. Osborn, and F. H. Molzahn, *J. Math. Phys.* **29**, 642 (1988).

[3] Y. Fujiwara, T. A. Osborn, and S. F. J. Wilk, *Phys. Rev. A.* **25**, 14 (1982).

[4] F. H. Molzahn, T. A. Osborn, *J. Phys. A.* **20**, 3073 (1987).

[5] S. A. Fulling, I. Borosh, *Cataloguing General Graphs by Point and Line Spectra.* (In Preparation).

[6] L. S. Brown, Y. Zhang, *Am. J. Phys.* **62**, 806 (1994).

[7] C. Farina, M. Maneschy, and C. Neves, *Am. J. Phys.* **61**, 636 (1993).

[8] J. Oddershede, P. Jørgensen, and D. L. Yeager, *Comp. Phys. Rep. 2.* 33 (1984).

[9] J. B. Delos, R. L. Waterland, and M. L. Du, *Phys. Rev. A.* **37**, 1185 (1988).

[10] R. L. Waterland, J. B. Delos, and M. L. Du, *Phys. Rev. A.* **35**, 5064 (1987).

[11] J. B. Delos, S. K. Knudson, and D. W. Noid, *Phys. Rev. A.* **28**, 7 (1983).

[12] N. Makri, W. H. Miller, *J. Chem. Phys.* **90**, 904 (1989).

[13] N. Makri, W. H. Miller, *Chem. Phys. L.* **151**, 1 (1988).

Appendix A

Formulas for Electromagnetic Field Expansion

Formulas for the Calculation of T_1 and T_2

The following formulas were extracted from Papiez, Osborn and Molzahn[2]. They were programmed in C (see Appendix B) for the calculation of T_1 and T_2.

$$g(\xi,\xi') = \xi_<(1-\xi_>)$$
$$f_i(x,t) = -(\nabla^i v)(x,t) - \partial a_i(x,t)$$
$$A_{ij}(x,t) = (\nabla^i a_j)(x,t) - (\nabla^j a_i)(x,t)$$

$\Omega_i(z,\tau) \equiv \Omega_i(z,\tau;Q) = f_i(z,\tau) + (t-s)^{-1}(x-y)_j A_{ij}(z,\tau)$,

where $i = 1 \sim d$ and (hereafter) the repeated index j is summed from 1 to d. Then

$$T_1(Q) = (2i\hbar)^{-1}(t-s)^3 \int_{I^2} d\xi_1\, d\xi_2\, g(\xi_1,\xi_2)\Omega_j(w(\xi_1))\Omega_j(w(\xi_2))$$
$$- (2)^{-1}(t-s)^2 \int_I d\xi_1(\xi_1 - \xi_1^2)\left[\nabla\cdot f(w(\xi_1)) + \left(\frac{x-y}{t-s}\right)_j \nabla^i A_{ij}(w(\xi_1))\right]. \tag{4.32}$$

Here g is the one-dimensional Green's function defined in (3.3e), and $I^n = [0,1]^n$. The function T_2 is more elaborate. One finds

$$T_2(Q) = \frac{1}{2}T_1(Q)^2 + \sum_{l=0}^{2} (i\hbar)^{l-1} G_l^2(Q), \tag{4.33a}$$

where the functions G_l are given by

$$G_0^2(Q) = -\frac{1}{2}(t-s)^5 \int_{I^3} d\xi_1\, d\xi_2\, d\xi_3\, g(\xi_1,\xi_2)\Omega_i(w(\xi_1))\Omega_j(w(\xi_3))$$
$$\times [g(\xi_2,\xi_3)\nabla^j\Omega_i(w(\xi_2)) + (t-s)^{-1}\partial_1 g(\xi_2,\xi_3)A_{ij}(w(\xi_2))], \tag{4.33b}$$

$$G_1^2(Q) = \frac{1}{4}(t-s)^4 \int_{I^2} d\xi_1\, d\xi_2 g(\xi_1,\xi_2)\{g(\xi_1,\xi_2)[\nabla^i\Omega_j(w(\xi_1))$$
$$+ (\xi_1(t-s))^{-1}F_{ji}(w(\xi_1))][\nabla^i\Omega_j(w(\xi_2)) + (\xi_2(t-s))^{-1}A_{ji}(w(\xi_2))]$$
$$+ \xi_2^2\xi_>^{-1}(1-\xi_<)\Omega_j(w(\xi_1))[\Delta\Omega_j(w(\xi_2)) + 2(\xi_2(t-s))^{-1}\nabla^i A_{ji}(w(\xi_2))]$$
$$+ \xi_2\xi_>^{-1}[2\xi_< - (\xi_< + 1)\xi_2]\Omega_j(w(\xi_1))[\nabla^j\nabla^i\Omega_i(w(\xi_2)) + (\xi_2(t-s))^{-1}\nabla^i A_{ij}(w(\xi_2))]\}, \tag{4.33c}$$

$$G_2^2(Q) = -(8)^{-1}(t-s)^3 \int_I d\xi_1[\xi_1(1-\xi_1)]^2[\Delta(\nabla\cdot\Omega)](w(\xi_1)). \tag{4.33d}$$

A-1

Numerical Differentiation

All of the numerical differentiation performed in *qsenumer.c* (the only place any numerical differentiation is performed) is done according to five-point numerical differentiation formulas. The five-point formula for the first derivative is commonly available in texts on numerical analysis.

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + \tfrac{1}{30} f^{(5)}(c)h^4$$

The five-point differentiation formula for the second derivative is not as commonly available, so I give its derivation here.

$$f(x+h) = f(x) + f'(x)h + \tfrac{1}{2} f''(x)h^2 + \tfrac{1}{6} f^{(3)}(x)h^3 + \tfrac{1}{24} f^{(4)}(x)h^4 + \tfrac{1}{120} f^{(5)}(x)h^5 + \tfrac{1}{720} f^{(6)}(c_1)h^6$$

where $c_1 \in (x, x+h)$

and similarly,

$$f(x-h) = f(x) - f'(x)h + \tfrac{1}{2} f''(x)h^2 - \tfrac{1}{6} f^{(3)}(x)h^3 + \tfrac{1}{24} f^{(4)}(x)h^4 - \tfrac{1}{120} f^{(5)}(x)h^5 + \tfrac{1}{720} f^{(6)}(c_2)h^6$$

where $c_2 \in (x-h, x)$.

$$\Rightarrow f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \tfrac{1}{12} f^{(4)}(x)h^4 - \tfrac{1}{360} f^{(6)}(c_3)h^4$$

where $c_3 \in (x-h, x+h)$

$$\Rightarrow f''(x) = \frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2} - \tfrac{1}{3} f^{(4)}(x)h^4 - \tfrac{2}{45} f^{(6)}(c_4)h^4$$

where $c_4 \in (x-2h, x+2h)$.

Combining, we get

$$3f''(x) = 4\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{1}{4}\frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2} - \tfrac{1}{90} f^{(6)}(c_3)h^4 + \tfrac{2}{45} f^{(6)}(c_4)h^4$$

and so

$$f''(x) = \frac{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)}{12h^2} + \tfrac{1}{90} f^{(6)}(c)h^4$$

The Evil Green's Function "g"

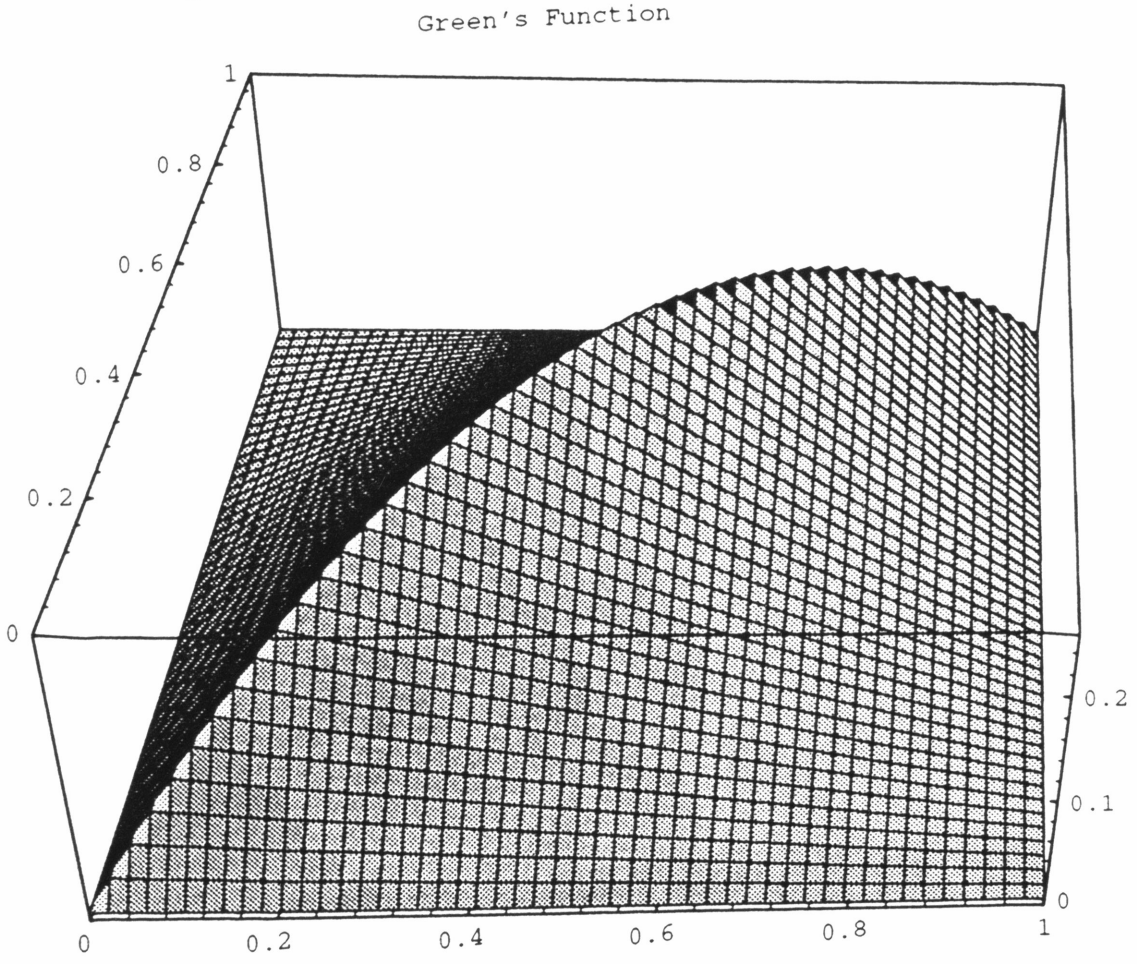
The green's function $g(\xi, \xi') = \xi_<(1 - \xi_>)$ that appears in the large mass expansion formulas (both the electromagnetic field expansion and the connected graph expansion formulas) causes some problems. As pictured on the following page, g has a discontinuous first derivative along the $\xi = \xi'$ line. First, conventional numerical integration schemes have difficulty integrating functions with discontinuous first derivatives. Second, Mathematica can not symbolically integrate g in its current formulation. Of course, there are ways around this.

The obvious solution for numerical integration is to effectively remove the trouble area. That is, if we are integrating $g(\xi_1,\xi_2)$ over the square ξ_1 from 0 to 1 and ξ_2 from 0 to 1, we cut the area into two triangles along the $\xi_1=\xi_2$ line and integrate separately over each triangle. The standard textbook Simpson's Rule can be applied effectively to these triangles and the numerical integration can proceed. In higher dimensions things do not work so easily. If we try to integrate $g(\xi_1,\xi_2)*g(\xi_2,\xi_3)*g(\xi_1,\xi_3)$ over the unit cube, we gain additional planes of discontinuous first derivatives that must be removed. A straight forward application of Simpson's Rule to these six simplices no longer integrates quadratics exactly as the one and two-dimensional implementations do. For even higher dimensions the difficulty persists. Thus for the triple numerical integration needed for the calculation of T_2 in the C code, we have developed two different integration schemes based upon the different basic quadrature rules of Simpson and Romberg. The two schemes differ in effectiveness depending upon the potentials chosen for the calculations. We have gained valuable insight on this matter from Dr. James Lyness and will continue investigating this non-trivial integration task.

To symbolically integrate $g(\xi_1,\xi_2)$ in Mathematica over the unit square, we must rewrite the integral as two integrals, one over each triangle (as explained earlier). For higher dimensional cases, integrals will need to be split multiple times. In the Mathematica code in Appendix B, this process is carried out by the routine *L*. However, in the connected graph expansion, this complicated splitting of integrals can be circumvented by moving the summation over connected graphs inside the integral. The sum over graphs is symmetric over all of the vertices of the n-dimensional cube to be integrated over. Thus we reduce the problem to an integration over a single simplex, which must then be multiplied by the number of simplices (n!). This simplification is employed in the routine *LExpSeries*.

☐ **Here we look at the Green's function "g" that is the troublemaker in our numerical integration attempts on T1.**

```
Plot3D[ g[kse1,kse2], {kse1,0,1}, {kse2,0,1},
        PlotPoints->50,
        PlotLabel->"Green's Function",
        ViewPoint->{0.3,-2.4,2},
        Lighting->True ]
```



Appendix B

Electromagnetic Field Expansion
C Source Code

```
/*
  flat.h  --  Include file for calculations in QSE paper.
              Contains prototypes for various functions and
              definitions for all structures used.

              Note:  To change the number of dimensions, change
              the line "#define d ?".  Replace the ? with the
              number of dimensions desired.

              Note:  To change a and v, modify the file qseinput.c
              appropriately.

              Note:  To change initial values x, y and t, modify
              the file containing main().

              Note:  If explicit versions of f, A, divf,
              partial derivatives of A, etc. are available, modify
              qsederiv.c appropriately.  If not, link with
              qsenumer.c instead.

                                                              */

#define d 3    /* Refers to the number of dimensions    */
#define h .1   /* Refers to the h used in the numerical
                  differentiation in qsenumer.c         */
#define SIMPSON 1   /* For use in specifying the integration scheme */
#define ROMBERG 2   /* to be used in the triple integration.        */

typedef struct
{
  double x[d];
  double y[d];
  double t;
} specpos;

typedef struct
{
  double vector[d];
  double scalar;
} position;

typedef struct
{
  double vector[d];
} vectform;
```

```
typedef struct
{
  double element[d][d];
} matrixform;

typedef struct
{
  double element[d][2];
} twovector;

typedef struct
{
  double ih_neg1;
  double ih_zero;
} T1form;

typedef struct
{
  double ih_neg2;
  double ih_neg1;
  double ih_zero;
  double ih_pos1;
} T2form;

/* qseinput.c */
double v( position XYZt );
vectform a( position XYZt );

/* qsefuncs.c */
double g( double kse1, double kse2 );
position w( double kse, specpos Q );
vectform Omega( position zetatau, specpos Q );
double max_of( double kse1, double kse2 );
double d1g( double kse2, double kse3 );

/* qsederiv.c or qsenumer.c */
vectform f( position zetatau );
matrixform MatrixA( position zetatau );
double divf( position zetatau );
matrixform GradOmega( position zetatau, specpos Q );
twovector GradA( position zetatau );
vectform LaplacianOmega( position zetatau, specpos Q );
double LapdivOmega( position zetatau, specpos Q );
vectform GraddivOmega( position zetatau, specpos Q );
```

```
/* qsetj.c */
T1form T1( specpos Q, int numofpts );
T2form T2( specpos Q, int numofpts, int tplintgscheme );
T1form J1( specpos Q, int numofpts );
T2form J2( specpos Q, int numofpts, int tplintgscheme );
double J( specpos Q, int numofpts );

/* numintg.c */
double Simp_Integrate( double (*Func)(double x, specpos Q), double xMin,
                       double xMax, int n, specpos Q );
double Simp_DblIntegrate( double (*Func)(double x, double y, specpos Q),
                          double xMin, double xMax, double yMin, double yMax,
                          int n, specpos Q );
double Simp_Simplex_TplIntegrate( double (*Func)(double x, double y, double z,
                                                 specpos Q),
                          double xMin, double xMax, double yMin, double yMax,
                          double zMin, double zMax, int n, specpos Q );
double Rmbg_Simplex_TplIntegrate( double (*Func)(double x, double y, double z,
                                                 specpos Q),
                          double xMin, double xMax, double yMin, double yMax,
                          double zMin, double zMax, int n, specpos Q );
```

```
/*
   numintg.c  -- Contains routines that integrate arbitrary functions
                 numerically using Simpson's integration formulas for
                 single and double integrals
              -- written by Davin M. Potts
              -- working version 21 June 1993
                                                                    */

#include "flat.h"


/*  Simp_Integrate, Simp_DblIntegrate, and Simp_TplIntegrate are supplied,
    as one argument, a pointer to the function that is to be integrated.
    This keeps the integrating functions versatile and general.        */


double Simp_Integrate( double (*Func)(double x, specpos Q), double xMin,
                       double xMax, int n, specpos Q )
{
  double even, odd;
  double incr = (xMax-xMin)/(double)n;
  double x;
  int index;

  even = 0;
  odd = 0;

  for( x = xMin + incr, index = 1; index < n; x += (2*incr), index += 2 )
    odd += (*Func)( x, Q );

  for( x = xMin + 2*incr, index = 2; index < n; x += (2*incr), index += 2 )
    even += (*Func)( x, Q );

  return( incr*( (*Func)(xMin,Q) + 4*odd + 2*even + (*Func)(xMax,Q) ) / 3 );
}

double Simp_DblIntegrate( double (*Func)(double x, double y, specpos Q),
                          double xMin, double xMax, double yMin, double yMax,
                          int n, specpos Q )
{
  double value;
  double xincr = (xMax-xMin)/(double)(2*n);
  double yincr = (yMax-yMin)/(double)(2*n);
  double x, y;
  int xindex, yindex;
```

```
  value = 0;

  /* Integrate the lower right triangle */
  for( xindex = 0, x = xMin; xindex <= (2*n); xindex++, x += xincr )
    for( yindex = 0, y = yMin; yindex <= xindex; yindex++, y += yincr )
      if( (xindex % 2 != 0) || (yindex % 2 != 0) )
        if( (yindex == 0) || (xindex == (2*n)) || (xindex == yindex) )
        {
          value += (double)2.0 * (*Func)( x, y, Q ) / (double)3.0;
        }
        else
        {
          value += (double)4.0 * (*Func)( x, y, Q ) / (double)3.0;
        }

  /* Integrate the upper left triangle */
  for( yindex = 0, y = yMin; yindex <= (2*n); yindex++, y += yincr )
    for( xindex = 0, x = xMin; xindex <= yindex; xindex++, x += xincr )
      if( (xindex % 2 != 0) || (yindex % 2 != 0) )
        if( (xindex == 0) || (yindex == (2*n)) || (xindex == yindex) )
        {
          value += (double)2.0 * (*Func)( x, y, Q ) / (double)3.0;
        }
        else
        {
          value += (double)4.0 * (*Func)( x, y, Q ) / (double)3.0;
        }

  value = value / ( 4*n*n );

  return( value );
}


double Simp_Simplex_TplIntegrate( double (*Func)(double x, double y, double z,
                                  specpos Q),
                                  double xMin, double xMax, double yMin, double yMax,
                                  double zMin, double zMax, int n, specpos Q )
{
  double value, weight;
  int numpts = 2*n;
  double xincr = (xMax-xMin)/(double)numpts;
  double yincr = (yMax-yMin)/(double)numpts;
  double zincr = (zMax-zMin)/(double)numpts;
```

```
double x, y, z;
int xindex, yindex, zindex, numodd;

value = 0;


/* Integrate the first tetrahedron - 0 < z < y < x < 1 */
for( xindex = 0, x = xMin; xindex <= (2*n); xindex++, x += xincr )
  for( yindex = 0, y = yMin; yindex <= xindex; yindex++, y += yincr )
    for( zindex = 0, z = zMin; zindex <= yindex; zindex++, z += zincr )
    {
      /* Count the number of odd indices. */
      numodd = 0;
      if( xindex % 2 == 1 ) numodd++;
      if( yindex % 2 == 1 ) numodd++;
      if( zindex % 2 == 1 ) numodd++;

      /* If on a FACE, this is special!  Weight is
        different from normal interior points.    */
      if( (xindex % numpts == 0) || (xindex==yindex) ||
         (yindex % numpts == 0) || (yindex==zindex) ||
         (zindex % numpts == 0) || (zindex==xindex) )
      {
        /* If on an EDGE, this is even more special! */
        if( ((xindex==yindex)&&(yindex==zindex)) ||
           ((xindex % numpts == 0)&&((yindex==zindex)||
            (yindex % numpts == 0)||(zindex % numpts == 0))) ||
           ((yindex % numpts == 0)&&((xindex==zindex)||
            (xindex % numpts == 0)||(zindex % numpts == 0))) ||
           ((zindex % numpts == 0)&&((xindex==yindex)||
            (xindex % numpts == 0)||(yindex % numpts == 0))) )
        {
          /* If on a VERTEX, change the weight to -1/15. */
          if( (xindex % numpts == 0) &&
              (yindex % numpts == 0) &&
              (zindex % numpts == 0) )
          { weight = (double)-1 / (double)15; }
          else
          {
            /* If on an EDGE ANY ODD, weight is 4/15. */
            if( numodd > 0 )
            { weight = (double)4 / (double)15; }
            else
            {
              /* If on a FACE-DIAGONAL EDGE, weight is -2/5. */
```

```
                if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
                    ((yindex % numpts == 0)&&(xindex==zindex)) ||
                    ((zindex % numpts == 0)&&(xindex==yindex)) )
                { weight = (double)-2 / (double)5; }
                /* If none of the above, we are at an OTHER EDGE,
                   so the weight is -4/15.                    */
                else
                { weight = (double)-4 / (double)15; }
              }
            }
          }
          else
          {
            /* We now know we are just on a FACE. */
            if( (numodd==1)||(numodd==3) )
            { weight = (double)4 / (double)5; }
            else
            {
              if( numodd==2 )
              { weight = (double)8 / (double)15; }
              else
              { weight = (double)-4 / (double)5; }
            }
          }
        }
        else
        {
          /* We are at an INTERIOR point. */
          if( (numodd==1)||(numodd==3) )
          { weight = (double)8 / (double)5; }
          else
          {
            if( numodd==2 )
            { weight = (double)16 / (double)15; }
            else
            { weight = (double)-8 / (double)5; }
          }
        }

        /* Sum the point with its weighting up with the rest. */
        value += weight * (*Func)( x, y, z, Q );
      }

/* Integrate the second tetrahedron - 0 < z < x < y < 1 */
for( yindex = 0, y = yMin; yindex <= (2*n); yindex++, y += yincr )
```

```
for( xindex = 0, x = xMin; xindex <= yindex; xindex++, x += xincr )
    for( zindex = 0, z = zMin; zindex <= xindex; zindex++, z += zincr )
  {
    /* Count the number of odd indices. */
    numodd = 0;
    if( xindex % 2 == 1 ) numodd++;
    if( yindex % 2 == 1 ) numodd++;
    if( zindex % 2 == 1 ) numodd++;

    /* If on a FACE, this is special!  Weight is
      different from normal interior points.    */
    if( (xindex % numpts == 0) || (xindex==yindex) ||
       (yindex % numpts == 0) || (yindex==zindex) ||
       (zindex % numpts == 0) || (zindex==xindex) )
    {
      /* If on an EDGE, this is even more special! */
      if( ((xindex==yindex)&&(yindex==zindex)) ||
          ((xindex % numpts == 0)&&((yindex==zindex)||
           (yindex % numpts == 0)||(zindex % numpts == 0))) ||
          ((yindex % numpts == 0)&&((xindex==zindex)||
           (xindex % numpts == 0)||(zindex % numpts == 0))) ||
          ((zindex % numpts == 0)&&((xindex==yindex)||
           (xindex % numpts == 0)||(yindex % numpts == 0))) )
      {
        /* If on a VERTEX, change the weight to -1/15. */
        if( (xindex % numpts == 0) &&
            (yindex % numpts == 0) &&
            (zindex % numpts == 0) )
        { weight = (double)-1 / (double)15; }
        else
        {
          /* If on an EDGE ANY ODD, weight is 4/15. */
          if( numodd > 0 )
          { weight = (double)4 / (double)15; }
          else
          {
            /* If on a FACE-DIAGONAL EDGE, weight is -2/5. */
            if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
                ((yindex % numpts == 0)&&(xindex==zindex)) ||
                ((zindex % numpts == 0)&&(xindex==yindex)) )
            { weight = (double)-2 / (double)5; }
            /* If none of the above, we are at an OTHER EDGE,
              so the weight is -4/15.                    */
            else
            { weight = (double)-4 / (double)15; }
```

```
            }
          }
        }
        else
        {
          /* We now know we are just on a FACE. */
          if( (numodd==1)||(numodd==3) )
          { weight = (double)4 / (double)5; }
          else
          {
            if( numodd==2 )
            { weight = (double)8 / (double)15; }
            else
            { weight = (double)-4 / (double)5; }
          }
        }
      }
      else
      {
        /* We are at an INTERIOR point. */
        if( (numodd==1)||(numodd==3) )
        { weight = (double)8 / (double)5; }
        else
        {
          if( numodd==2 )
          { weight = (double)16 / (double)15; }
          else
          { weight = (double)-8 / (double)5; }
        }
      }

      /* Sum the point with its weighting up with the rest. */
      value += weight * (*Func)( x, y, z, Q );
    }

/* Integrate the third tetrahedron - 0 < x < z < y < 1 */
for( yindex = 0, y = yMin; yindex <= (2*n); yindex++, y += yincr )
  for( zindex = 0, z = zMin; zindex <= yindex; zindex++, z += zincr )
      for( xindex = 0, x = xMin; xindex <= zindex; xindex++, x += xincr )
    {
      /* Count the number of odd indices. */
      numodd = 0;
      if( xindex % 2 == 1 ) numodd++;
      if( yindex % 2 == 1 ) numodd++;
      if( zindex % 2 == 1 ) numodd++;
```

```
/* If on a FACE, this is special!  Weight is
   different from normal interior points.    */
if( (xindex % numpts == 0) || (xindex==yindex) ||
    (yindex % numpts == 0) || (yindex==zindex) ||
    (zindex % numpts == 0) || (zindex==xindex) )
{
  /* If on an EDGE, this is even more special! */
  if( ((xindex==yindex)&&(yindex==zindex)) ||
      ((xindex % numpts == 0)&&((yindex==zindex)||
       (yindex % numpts == 0)||(zindex % numpts == 0))) ||
      ((yindex % numpts == 0)&&((xindex==zindex)||
       (xindex % numpts == 0)||(zindex % numpts == 0))) ||
      ((zindex % numpts == 0)&&((xindex==yindex)||
       (xindex % numpts == 0)||(yindex % numpts == 0))) )
  {
    /* If on a VERTEX, change the weight to -1/15. */
    if( (xindex % numpts == 0) &&
        (yindex % numpts == 0) &&
        (zindex % numpts == 0) )
    { weight = (double)-1 / (double)15; }
    else
    {
      /* If on an EDGE ANY ODD, weight is 4/15. */
      if( numodd > 0 )
      { weight = (double)4 / (double)15; }
      else
      {
        /* If on a FACE-DIAGONAL EDGE, weight is -2/5. */
        if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
            ((yindex % numpts == 0)&&(xindex==zindex)) ||
            ((zindex % numpts == 0)&&(xindex==yindex)) )
        { weight = (double)-2 / (double)5; }
        /* If none of the above, we are at an OTHER EDGE,
           so the weight is -4/15.                    */
        else
        { weight = (double)-4 / (double)15; }
      }
    }
  }
  else
  {
    /* We now know we are just on a FACE. */
    if( (numodd==1)||(numodd==3) )
    { weight = (double)4 / (double)5; }
```

```
            else
            {
              if( numodd==2 )
              { weight = (double)8 / (double)15; }
              else
              { weight = (double)-4 / (double)5; }
            }
          }
        }
        else
        {
          /* We are at an INTERIOR point. */
          if( (numodd==1)||(numodd==3) )
          { weight = (double)8 / (double)5; }
          else
          {
            if( numodd==2 )
            { weight = (double)16 / (double)15; }
            else
            { weight = (double)-8 / (double)5; }
          }
        }

        /* Sum the point with its weighting up with the rest. */
        value += weight * (*Func)( x, y, z, Q );
      }

/* Integrate the fourth tetrahedron - 0 < x < y < z < 1 */
for( zindex = 0, z = zMin; zindex <= (2*n); zindex++, z += zincr )
  for( yindex = 0, y = yMin; yindex <= zindex; yindex++, y += yincr )
      for( xindex = 0, x = xMin; xindex <= yindex; xindex++, x += xincr )
    {
      /* Count the number of odd indices. */
      numodd = 0;
      if( xindex % 2 == 1 ) numodd++;
      if( yindex % 2 == 1 ) numodd++;
      if( zindex % 2 == 1 ) numodd++;

      /* If on a FACE, this is special!  Weight is
        different from normal interior points.    */
      if( (xindex % numpts == 0) || (xindex==yindex) ||
        (yindex % numpts == 0) || (yindex==zindex) ||
        (zindex % numpts == 0) || (zindex==xindex) )
      {
        /* If on an EDGE, this is even more special! */
```

```
if( ((xindex==yindex)&&(yindex==zindex)) ||
   ((xindex % numpts == 0)&&((yindex==zindex)||
    (yindex % numpts == 0)||(zindex % numpts == 0))) ||
   ((yindex % numpts == 0)&&((xindex==zindex)||
    (xindex % numpts == 0)||(zindex % numpts == 0))) ||
   ((zindex % numpts == 0)&&((xindex==yindex)||
    (xindex % numpts == 0)||(yindex % numpts == 0))) )
{
  /* If on a VERTEX, change the weight to -1/15. */
  if( (xindex % numpts == 0) &&
      (yindex % numpts == 0) &&
      (zindex % numpts == 0) )
  { weight = (double)-1 / (double)15; }
  else
  {
    /* If on an EDGE ANY ODD, weight is 4/15. */
    if( numodd > 0 )
    { weight = (double)4 / (double)15; }
    else
    {
      /* If on a FACE-DIAGONAL EDGE, weight is -2/5. */
      if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
          ((yindex % numpts == 0)&&(xindex==zindex)) ||
          ((zindex % numpts == 0)&&(xindex==yindex)) )
      { weight = (double)-2 / (double)5; }
      /* If none of the above, we are at an OTHER EDGE,
        so the weight is -4/15.                    */
      else
      { weight = (double)-4 / (double)15; }
    }
  }
}
else
{
  /* We now know we are just on a FACE. */
  if( (numodd==1)||(numodd==3) )
  { weight = (double)4 / (double)5; }
  else
  {
    if( numodd==2 )
    { weight = (double)8 / (double)15; }
    else
    { weight = (double)-4 / (double)5; }
  }
}
```

```
        }
        else
        {
          /* We are at an INTERIOR point. */
          if( (numodd==1)||(numodd==3) )
          { weight = (double)8 / (double)5; }
          else
          {
            if( numodd==2 )
            { weight = (double)16 / (double)15; }
            else
            { weight = (double)-8 / (double)5; }
          }
        }

        /* Sum the point with its weighting up with the rest. */
        value += weight * (*Func)( x, y, z, Q );
      }

/* Integrate the fifth tetrahedron - 0 < y < x < z < 1 */
for( zindex = 0, z = zMin; zindex <= (2*n); zindex++, z += zincr )
  for( xindex = 0, x = xMin; xindex <= zindex; xindex++, x += xincr )
        for( yindex = 0, y = yMin; yindex <= xindex; yindex++, y += yincr )
        {
        /* Count the number of odd indices. */
        numodd = 0;
        if( xindex % 2 == 1 ) numodd++;
        if( yindex % 2 == 1 ) numodd++;
        if( zindex % 2 == 1 ) numodd++;

        /* If on a FACE, this is special!  Weight is
          different from normal interior points.    */
        if( (xindex % numpts == 0) || (xindex==yindex) ||
           (yindex % numpts == 0) || (yindex==zindex) ||
           (zindex % numpts == 0) || (zindex==xindex) )
        {
          /* If on an EDGE, this is even more special! */
          if( ((xindex==yindex)&&(yindex==zindex)) ||
             ((xindex % numpts == 0)&&((yindex==zindex)||
              (yindex % numpts == 0)||(zindex % numpts == 0))) ||
             ((yindex % numpts == 0)&&((xindex==zindex)||
              (xindex % numpts == 0)||(zindex % numpts == 0))) ||
             ((zindex % numpts == 0)&&((xindex==yindex)||
              (xindex % numpts == 0)||(yindex % numpts == 0))) )
          {
```

```
      /* If on a VERTEX, change the weight to -1/15. */
      if( (xindex % numpts == 0) &&
          (yindex % numpts == 0) &&
          (zindex % numpts == 0) )
      { weight = (double)-1 / (double)15; }
      else
      {
        /* If on an EDGE ANY ODD, weight is 4/15. */
        if( numodd > 0 )
        { weight = (double)4 / (double)15; }
        else
        {
          /* If on a FACE-DIAGONAL EDGE, weight is -2/5. */
          if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
              ((yindex % numpts == 0)&&(xindex==zindex)) ||
              ((zindex % numpts == 0)&&(xindex==yindex)) )
          { weight = (double)-2 / (double)5; }
          /* If none of the above, we are at an OTHER EDGE,
             so the weight is -4/15.                    */
          else
          { weight = (double)-4 / (double)15; }
        }
      }
    }
    else
    {
      /* We now know we are just on a FACE. */
      if( (numodd==1)||(numodd==3) )
      { weight = (double)4 / (double)5; }
      else
      {
        if( numodd==2 )
        { weight = (double)8 / (double)15; }
        else
        { weight = (double)-4 / (double)5; }
      }
    }
  }
  else
  {
    /* We are at an INTERIOR point. */
    if( (numodd==1)||(numodd==3) )
    { weight = (double)8 / (double)5; }
    else
    {
```

```
            if( numodd==2 )
            { weight = (double)16 / (double)15; }
            else
            { weight = (double)-8 / (double)5; }
          }
        }

        /* Sum the point with its weighting up with the rest. */
        value += weight * (*Func)( x, y, z, Q );
      }

/* Integrate the sixth tetrahedron - 0 < y < z < x < 1 */
for( xindex = 0, x = xMin; xindex <= (2*n); xindex++, x += xincr )
  for( zindex = 0, z = zMin; zindex <= xindex; zindex++, z += zincr )
      for( yindex = 0, y = yMin; yindex <= zindex; yindex++, y += yincr )
    {
      /* Count the number of odd indices. */
      numodd = 0;
      if( xindex % 2 == 1 ) numodd++;
      if( yindex % 2 == 1 ) numodd++;
      if( zindex % 2 == 1 ) numodd++;

      /* If on a FACE, this is special!  Weight is
        different from normal interior points.    */
      if( (xindex % numpts == 0) || (xindex==yindex) ||
        (yindex % numpts == 0) || (yindex==zindex) ||
        (zindex % numpts == 0) || (zindex==xindex) )
      {
        /* If on an EDGE, this is even more special! */
        if( ((xindex==yindex)&&(yindex==zindex)) ||
          ((xindex % numpts == 0)&&((yindex==zindex)||
           (yindex % numpts == 0)||(zindex % numpts == 0))) ||
          ((yindex % numpts == 0)&&((xindex==zindex)||
           (xindex % numpts == 0)||(zindex % numpts == 0))) ||
          ((zindex % numpts == 0)&&((xindex==yindex)||
           (xindex % numpts == 0)||(yindex % numpts == 0))) )
        {
          /* If on a VERTEX, change the weight to -1/15. */
          if( (xindex % numpts == 0) &&
            (yindex % numpts == 0) &&
            (zindex % numpts == 0) )
          { weight = (double)-1 / (double)15; }
          else
          {
            /* If on an EDGE ANY ODD, weight is 4/15. */
```

```
        if( numodd > 0 )
        { weight = (double)4 / (double)15; }
        else
        {
          /* If on a FACE-DIAGONAL EDGE, weight is -2/5. */
          if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
              ((yindex % numpts == 0)&&(xindex==zindex)) ||
              ((zindex % numpts == 0)&&(xindex==yindex)) )
          { weight = (double)-2 / (double)5; }
          /* If none of the above, we are at an OTHER EDGE,
             so the weight is -4/15.                    */
          else
          { weight = (double)-4 / (double)15; }
        }
      }
    }
    else
    {
      /* We now know we are just on a FACE. */
      if( (numodd==1)||(numodd==3) )
      { weight = (double)4 / (double)5; }
      else
      {
        if( numodd==2 )
        { weight = (double)8 / (double)15; }
        else
        { weight = (double)-4 / (double)5; }
      }
    }
  }
  else
  {
    /* We are at an INTERIOR point. */
    if( (numodd==1)||(numodd==3) )
    { weight = (double)8 / (double)5; }
    else
    {
      if( numodd==2 )
      { weight = (double)16 / (double)15; }
      else
      { weight = (double)-8 / (double)5; }
    }
  }

  /* Sum the point with its weighting up with the rest. */
```

```
        if( numodd > 0 )
        { weight = (double)4 / (double)15; }
        else
        {
          /* If on a FACE-DIAGONAL EDGE, weight is -2/5. */
          if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
              ((yindex % numpts == 0)&&(xindex==zindex)) ||
              ((zindex % numpts == 0)&&(xindex==yindex)) )
          { weight = (double)-2 / (double)5; }
          /* If none of the above, we are at an OTHER EDGE,
             so the weight is -4/15.                    */
          else
          { weight = (double)-4 / (double)15; }
        }
      }
    }
    else
    {
      /* We now know we are just on a FACE. */
      if( (numodd==1)||(numodd==3) )
      { weight = (double)4 / (double)5; }
      else
      {
        if( numodd==2 )
        { weight = (double)8 / (double)15; }
        else
        { weight = (double)-4 / (double)5; }
      }
    }
  }
  else
  {
    /* We are at an INTERIOR point. */
    if( (numodd==1)||(numodd==3) )
    { weight = (double)8 / (double)5; }
    else
    {
      if( numodd==2 )
      { weight = (double)16 / (double)15; }
      else
      { weight = (double)-8 / (double)5; }
    }
  }

  /* Sum the point with its weighting up with the rest. */
```

```
        value += weight * (*Func)( x, y, z, Q );
      }

  /* Divide by the number of points integrated over. */
  value = value / ( (double)numpts * (double)numpts * (double)numpts );

  return( value );
}

double Rmbg_Simplex_TplIntegrate( double (*Func)(double x, double y, double z,
                                  specpos Q),
                                  double xMin, double xMax, double yMin, double yMax,
                                  double zMin, double zMax, int n, specpos Q )
{
  double value, weight;
  int numpts = 2*n;
  double xincr = (xMax-xMin)/(double)numpts;
  double yincr = (yMax-yMin)/(double)numpts;
  double zincr = (zMax-zMin)/(double)numpts;
  double x, y, z;
  int xindex, yindex, zindex, numodd;

  value = 0;


  /* Integrate the first tetrahedron - 0 < z < y < x < 1 */
  for( xindex = 0, x = xMin; xindex <= (2*n); xindex++, x += xincr )
    for( yindex = 0, y = yMin; yindex <= xindex; yindex++, y += yincr )
      for( zindex = 0, z = zMin; zindex <= yindex; zindex++, z += zincr )
      {
        /* Count the number of odd indices. */
        numodd = 0;
        if( xindex % 2 == 1 ) numodd++;
        if( yindex % 2 == 1 ) numodd++;
        if( zindex % 2 == 1 ) numodd++;

        /* If on a FACE, this is special!  Weight is
           different from normal interior points.    */
        if( (xindex % numpts == 0) || (xindex==yindex) ||
            (yindex % numpts == 0) || (yindex==zindex) ||
            (zindex % numpts == 0) || (zindex==xindex) )
        {
          /* If on an EDGE, this is even more special! */
          if( ((xindex==yindex)&&(yindex==zindex)) ||
              ((xindex % numpts == 0)&&((yindex==zindex)||
```

```
     (yindex % numpts == 0)||(zindex % numpts == 0))) ||
    ((yindex % numpts == 0)&&((xindex==zindex)||
     (xindex % numpts == 0)||(zindex % numpts == 0))) ||
    ((zindex % numpts == 0)&&((xindex==yindex)||
     (xindex % numpts == 0)||(yindex % numpts == 0))) )
 {
   /* If on a VERTEX, change the weight to -1/18. */
   if( (xindex % numpts == 0) &&
       (yindex % numpts == 0) &&
       (zindex % numpts == 0) )
   { weight = (double)-1 / (double)18; }
   else
   {
     /* If on a FACE-DIAGONAL EDGE, ... */
     if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
         ((yindex % numpts == 0)&&(xindex==zindex)) ||
         ((zindex % numpts == 0)&&(xindex==yindex)) )
     {
       if( numodd > 0 )
       { weight = (double)1 / (double)3; }
       else
       { weight = (double)-1 / (double)3; }
     }
     /* If none of the above, we are at an OTHER EDGE. */
     else
     {
       if( numodd > 0 )
       { weight = (double)2 / (double)9; }
       else
       { weight = (double)-2 / (double)9; }
     }
   }
 }
 else
 {
   /* We now know we are just on a FACE. */
   /* If on a FACE ANY ODD, weight is 2/3. */
   if( numodd > 0 )
   { weight = (double)2 / (double)3; }
   else
   {
     /* We are on a FACE ALL EVEN. */
     weight = (double)-2 / (double)3;
   }
 }
```

```
        }
        else
        {
          /* We are at an INTERIOR point. */
          /* If on an INTERIOR ANY ODD, weight is 4/3. */
          if( numodd > 0 )
          { weight = (double)4 / (double)3; }
          else
          {
            /* We are at an INTERIOR ALL EVEN. */
            weight = (double)-4 / (double)3;
          }
        }

        /* Sum the point with its weighting up with the rest. */
        value += weight * (*Func)( x, y, z, Q );
      }

/* Integrate the second tetrahedron - 0 < z < x < y < 1 */
for( yindex = 0, y = yMin; yindex <= (2*n); yindex++, y += yincr )
  for( xindex = 0, x = xMin; xindex <= yindex; xindex++, x += xincr )
      for( zindex = 0, z = zMin; zindex <= xindex; zindex++, z += zincr )
    {
      /* Count the number of odd indices. */
      numodd = 0;
      if( xindex % 2 == 1 ) numodd++;
      if( yindex % 2 == 1 ) numodd++;
      if( zindex % 2 == 1 ) numodd++;

      /* If on a FACE, this is special!  Weight is
        different from normal interior points.    */
      if( (xindex % numpts == 0) || (xindex==yindex) ||
         (yindex % numpts == 0) || (yindex==zindex) ||
         (zindex % numpts == 0) || (zindex==xindex) )
      {
        /* If on an EDGE, this is even more special! */
        if( ((xindex==yindex)&&(yindex==zindex)) ||
           ((xindex % numpts == 0)&&((yindex==zindex)||
            (yindex % numpts == 0)||(zindex % numpts == 0))) ||
           ((yindex % numpts == 0)&&((xindex==zindex)||
            (xindex % numpts == 0)||(zindex % numpts == 0))) ||
           ((zindex % numpts == 0)&&((xindex==yindex)||
            (xindex % numpts == 0)||(yindex % numpts == 0))) )
        {
          /* If on a VERTEX, change the weight to -1/18. */
```

```
    if( (xindex % numpts == 0) &&
        (yindex % numpts == 0) &&
        (zindex % numpts == 0) )
    { weight = (double)-1 / (double)18; }
    else
    {
      /* If on a FACE-DIAGONAL EDGE, ... */
      if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
          ((yindex % numpts == 0)&&(xindex==zindex)) ||
          ((zindex % numpts == 0)&&(xindex==yindex)) )
      {
        if( numodd > 0 )
        { weight = (double)1 / (double)3; }
        else
        { weight = (double)-1 / (double)3; }
      }
      /* If none of the above, we are at an OTHER EDGE. */
      else
      {
        if( numodd > 0 )
        { weight = (double)2 / (double)9; }
        else
        { weight = (double)-2 / (double)9; }
      }
    }
  }
  else
  {
    /* We now know we are just on a FACE. */
    /* If on a FACE ANY ODD, weight is 2/3. */
    if( numodd > 0 )
    { weight = (double)2 / (double)3; }
    else
    {
      /* We are on a FACE ALL EVEN. */
      weight = (double)-2 / (double)3;
    }
  }
}
else
{
  /* We are at an INTERIOR point. */
  /* If on an INTERIOR ANY ODD, weight is 4/3. */
  if( numodd > 0 )
  { weight = (double)4 / (double)3; }
```

```
          else
          {
            /* We are at an INTERIOR ALL EVEN. */
            weight = (double)-4 / (double)3;
          }
        }

        /* Sum the point with its weighting up with the rest. */
        value += weight * (*Func)( x, y, z, Q );
      }

/* Integrate the third tetrahedron - 0 < x < z < y < 1 */
for( yindex = 0, y = yMin; yindex <= (2*n); yindex++, y += yincr )
  for( zindex = 0, z = zMin; zindex <= yindex; zindex++, z += zincr )
      for( xindex = 0, x = xMin; xindex <= zindex; xindex++, x += xincr )
    {
      /* Count the number of odd indices. */
      numodd = 0;
      if( xindex % 2 == 1 ) numodd++;
      if( yindex % 2 == 1 ) numodd++;
      if( zindex % 2 == 1 ) numodd++;

      /* If on a FACE, this is special!  Weight is
        different from normal interior points.    */
      if( (xindex % numpts == 0) || (xindex==yindex) ||
         (yindex % numpts == 0) || (yindex==zindex) ||
         (zindex % numpts == 0) || (zindex==xindex) )
      {
        /* If on an EDGE, this is even more special! */
        if( ((xindex==yindex)&&(yindex==zindex)) ||
           ((xindex % numpts == 0)&&((yindex==zindex)||
            (yindex % numpts == 0)||(zindex % numpts == 0))) ||
           ((yindex % numpts == 0)&&((xindex==zindex)||
            (xindex % numpts == 0)||(zindex % numpts == 0))) ||
           ((zindex % numpts == 0)&&((xindex==yindex)||
            (xindex % numpts == 0)||(yindex % numpts == 0))) )
        {
          /* If on a VERTEX, change the weight to -1/18. */
          if( (xindex % numpts == 0) &&
             (yindex % numpts == 0) &&
             (zindex % numpts == 0) )
          { weight = (double)-1 / (double)18; }
          else
          {
            /* If on a FACE-DIAGONAL EDGE, ... */
```

```
        if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
            ((yindex % numpts == 0)&&(xindex==zindex)) ||
            ((zindex % numpts == 0)&&(xindex==yindex)) )
        {
          if( numodd > 0 )
          { weight = (double)1 / (double)3; }
          else
          { weight = (double)-1 / (double)3; }
        }
        /* If none of the above, we are at an OTHER EDGE. */
        else
        {
          if( numodd > 0 )
          { weight = (double)2 / (double)9; }
          else
          { weight = (double)-2 / (double)9; }
        }
      }
    }
    else
    {
      /* We now know we are just on a FACE. */
      /* If on a FACE ANY ODD, weight is 2/3. */
      if( numodd > 0 )
      { weight = (double)2 / (double)3; }
      else
      {
        /* We are on a FACE ALL EVEN. */
        weight = (double)-2 / (double)3;
      }
    }
  }
  else
  {
    /* We are at an INTERIOR point. */
    /* If on an INTERIOR ANY ODD, weight is 4/3. */
    if( numodd > 0 )
    { weight = (double)4 / (double)3; }
    else
    {
      /* We are at an INTERIOR ALL EVEN. */
      weight = (double)-4 / (double)3;
    }
  }
```

```
      /* Sum the point with its weighting up with the rest. */
      value += weight * (*Func)( x, y, z, Q );
    }

/* Integrate the fourth tetrahedron - 0 < x < y < z < 1 */
for( zindex = 0, z = zMin; zindex <= (2*n); zindex++, z += zincr )
  for( yindex = 0, y = yMin; yindex <= zindex; yindex++, y += yincr )
      for( xindex = 0, x = xMin; xindex <= yindex; xindex++, x += xincr )
    {
      /* Count the number of odd indices. */
      numodd = 0;
      if( xindex % 2 == 1 ) numodd++;
      if( yindex % 2 == 1 ) numodd++;
      if( zindex % 2 == 1 ) numodd++;

      /* If on a FACE, this is special!  Weight is
        different from normal interior points.    */
      if( (xindex % numpts == 0) || (xindex==yindex) ||
         (yindex % numpts == 0) || (yindex==zindex) ||
         (zindex % numpts == 0) || (zindex==xindex) )
      {
        /* If on an EDGE, this is even more special! */
        if( ((xindex==yindex)&&(yindex==zindex)) ||
           ((xindex % numpts == 0)&&((yindex==zindex)||
            (yindex % numpts == 0)||(zindex % numpts == 0))) ||
           ((yindex % numpts == 0)&&((xindex==zindex)||
            (xindex % numpts == 0)||(zindex % numpts == 0))) ||
           ((zindex % numpts == 0)&&((xindex==yindex)||
            (xindex % numpts == 0)||(yindex % numpts == 0))) )
        {
          /* If on a VERTEX, change the weight to -1/18. */
          if( (xindex % numpts == 0) &&
             (yindex % numpts == 0) &&
             (zindex % numpts == 0) )
          { weight = (double)-1 / (double)18; }
          else
          {
            /* If on a FACE-DIAGONAL EDGE, ... */
            if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
               ((yindex % numpts == 0)&&(xindex==zindex)) ||
               ((zindex % numpts == 0)&&(xindex==yindex)) )
            {
              if( numodd > 0 )
              { weight = (double)1 / (double)3; }
              else
```

```
            { weight = (double)-1 / (double)3; }
          }
          /* If none of the above, we are at an OTHER EDGE. */
          else
          {
            if( numodd > 0 )
            { weight = (double)2 / (double)9; }
            else
            { weight = (double)-2 / (double)9; }
          }
        }
      }
      else
      {
        /* We now know we are just on a FACE. */
        /* If on a FACE ANY ODD, weight is 2/3. */
        if( numodd > 0 )
        { weight = (double)2 / (double)3; }
        else
        {
          /* We are on a FACE ALL EVEN. */
          weight = (double)-2 / (double)3;
        }
      }
    }
    else
    {
      /* We are at an INTERIOR point. */
      /* If on an INTERIOR ANY ODD, weight is 4/3. */
      if( numodd > 0 )
      { weight = (double)4 / (double)3; }
      else
      {
        /* We are at an INTERIOR ALL EVEN. */
        weight = (double)-4 / (double)3;
      }
    }

    /* Sum the point with its weighting up with the rest. */
    value += weight * (*Func)( x, y, z, Q );
  }

/* Integrate the fifth tetrahedron - 0 < y < x < z < 1 */
for( zindex = 0, z = zMin; zindex <= (2*n); zindex++, z += zincr )
  for( xindex = 0, x = xMin; xindex <= zindex; xindex++, x += xincr )
```

```
for( yindex = 0, y = yMin; yindex <= xindex; yindex++, y += yincr )
{
  /* Count the number of odd indices. */
  numodd = 0;
  if( xindex % 2 == 1 ) numodd++;
  if( yindex % 2 == 1 ) numodd++;
  if( zindex % 2 == 1 ) numodd++;

/* If on a FACE, this is special!  Weight is
  different from normal interior points.    */
if( (xindex % numpts == 0) || (xindex==yindex) ||
   (yindex % numpts == 0) || (yindex==zindex) ||
   (zindex % numpts == 0) || (zindex==xindex) )
{
  /* If on an EDGE, this is even more special! */
  if( ((xindex==yindex)&&(yindex==zindex)) ||
     ((xindex % numpts == 0)&&((yindex==zindex)||
      (yindex % numpts == 0)||(zindex % numpts == 0))) ||
     ((yindex % numpts == 0)&&((xindex==zindex)||
      (xindex % numpts == 0)||(zindex % numpts == 0))) ||
     ((zindex % numpts == 0)&&((xindex==yindex)||
      (xindex % numpts == 0)||(yindex % numpts == 0))) )
  {
    /* If on a VERTEX, change the weight to -1/18. */
    if( (xindex % numpts == 0) &&
       (yindex % numpts == 0) &&
       (zindex % numpts == 0) )
    { weight = (double)-1 / (double)18; }
    else
    {
      /* If on a FACE-DIAGONAL EDGE, ... */
      if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
         ((yindex % numpts == 0)&&(xindex==zindex)) ||
         ((zindex % numpts == 0)&&(xindex==yindex)) )
      {
        if( numodd > 0 )
        { weight = (double)1 / (double)3; }
        else
        { weight = (double)-1 / (double)3; }
      }
      /* If none of the above, we are at an OTHER EDGE. */
      else
      {
        if( numodd > 0 )
        { weight = (double)2 / (double)9; }
```

```
                else
                { weight = (double)-2 / (double)9; }
              }
            }
          }
          else
          {
            /* We now know we are just on a FACE. */
            /* If on a FACE ANY ODD, weight is 2/3. */
            if( numodd > 0 )
            { weight = (double)2 / (double)3; }
            else
            {
              /* We are on a FACE ALL EVEN. */
              weight = (double)-2 / (double)3;
            }
          }
        }
        else
        {
          /* We are at an INTERIOR point. */
          /* If on an INTERIOR ANY ODD, weight is 4/3. */
          if( numodd > 0 )
          { weight = (double)4 / (double)3; }
          else
          {
            /* We are at an INTERIOR ALL EVEN. */
            weight = (double)-4 / (double)3;
          }
        }

        /* Sum the point with its weighting up with the rest. */
        value += weight * (*Func)( x, y, z, Q );
      }

/* Integrate the sixth tetrahedron - 0 < y < z < x < 1 */
for( xindex = 0, x = xMin; xindex <= (2*n); xindex++, x += xincr )
  for( zindex = 0, z = zMin; zindex <= xindex; zindex++, z += zincr )
    for( yindex = 0, y = yMin; yindex <= zindex; yindex++, y += yincr )
    {
      /* Count the number of odd indices. */
      numodd = 0;
      if( xindex % 2 == 1 ) numodd++;
      if( yindex % 2 == 1 ) numodd++;
      if( zindex % 2 == 1 ) numodd++;
```

```
/* If on a FACE, this is special!  Weight is
  different from normal interior points.    */
if( (xindex % numpts == 0) || (xindex==yindex) ||
   (yindex % numpts == 0) || (yindex==zindex) ||
   (zindex % numpts == 0) || (zindex==xindex) )
{
  /* If on an EDGE, this is even more special! */
  if( ((xindex==yindex)&&(yindex==zindex)) ||
     ((xindex % numpts == 0)&&((yindex==zindex)||
      (yindex % numpts == 0)||(zindex % numpts == 0))) ||
     ((yindex % numpts == 0)&&((xindex==zindex)||
      (xindex % numpts == 0)||(zindex % numpts == 0))) ||
     ((zindex % numpts == 0)&&((xindex==yindex)||
      (xindex % numpts == 0)||(yindex % numpts == 0))) )
  {
    /* If on a VERTEX, change the weight to -1/18. */
    if( (xindex % numpts == 0) &&
        (yindex % numpts == 0) &&
        (zindex % numpts == 0) )
    { weight = (double)-1 / (double)18; }
    else
    {
      /* If on a FACE-DIAGONAL EDGE, ... */
      if( ((xindex % numpts == 0)&&(yindex==zindex)) ||
          ((yindex % numpts == 0)&&(xindex==zindex)) ||
          ((zindex % numpts == 0)&&(xindex==yindex)) )
      {
        if( numodd > 0 )
        { weight = (double)1 / (double)3; }
        else
        { weight = (double)-1 / (double)3; }
      }
      /* If none of the above, we are at an OTHER EDGE. */
      else
      {
        if( numodd > 0 )
        { weight = (double)2 / (double)9; }
        else
        { weight = (double)-2 / (double)9; }
      }
    }
  }
  else
  {
```

```
            /* We now know we are just on a FACE. */
            /* If on a FACE ANY ODD, weight is 2/3. */
            if( numodd > 0 )
            { weight = (double)2 / (double)3; }
            else
            {
              /* We are on a FACE ALL EVEN. */
              weight = (double)-2 / (double)3;
            }
          }
        }
        else
        {
          /* We are at an INTERIOR point. */
          /* If on an INTERIOR ANY ODD, weight is 4/3. */
          if( numodd > 0 )
          { weight = (double)4 / (double)3; }
          else
          {
            /* We are at an INTERIOR ALL EVEN. */
            weight = (double)-4 / (double)3;
          }
        }

        /* Sum the point with its weighting up with the rest. */
        value += weight * (*Func)( x, y, z, Q );
      }

  /* Divide by the number of points integrated over. */
  value = value / ( (double)numpts * (double)numpts * (double)numpts );

  return( value );
}
```

```
/*
  qsederiv.c  --  Explicit definitions for f, MatrixA, divf,
                  GradOmega, GradA, LaplacianOmega,
              LapdivOmega, and GraddivOmega.
              Meant to replace qsenumer.c.  Use this file
              when explicit expressions for these functions
              are available.  When they are not available,
                  do not link with this file - use the file
                  qsenumer.c instead.
                                                          */
#include "flat.h"
#include <math.h>

vectform f( position zetatau )
{
  /*  zetatau.vector[i] refers to the (i+1)th direction
      e.g.  zetatau.vector[0] = x direction
            zetatau.vector[1] = y direction  in 3 dimensional space
            zetatau.vector[2] = z direction                  */

  vectform value;

  value.vector[0] = -2 * zetatau.vector[0];
  value.vector[1] = 0;
  value.vector[2] = 0;

  return( value );
}

matrixform MatrixA( position zetatau )
{
  /*  zetatau.vector[i] refers to the (i+1)th direction
      e.g.  zetatau.vector[0] = x direction
            zetatau.vector[1] = y direction  in 3 dimensional space
            zetatau.vector[2] = z direction                  */

  matrixform value;

  value.element[0][0] = 0;
  value.element[0][1] = 0;
  value.element[0][2] = 0;
  value.element[1][0] = 0;
  value.element[1][1] = 0;
  value.element[1][2] = 0;
  value.element[2][0] = 0;
```

```
  value.element[2][1] = 0;
  value.element[2][2] = 0;

  return( value );
}

double divf( position zetatau )
{
  /*  zetatau.vector[i] refers to the (i+1)th direction
      e.g.  zetatau.vector[0] = x direction
            zetatau.vector[1] = y direction  in 3 dimensional space
            zetatau.vector[2] = z direction                    */

  double value;

  value = -2;

  return( value );
}

matrixform GradOmega( position zetatau, specpos Q )
{
  matrixform value;

  value.element[0][0] = -2;
  value.element[0][1] = 0;
  value.element[0][2] = 0;
  value.element[1][0] = 0;
  value.element[1][1] = 0;
  value.element[1][2] = 0;
  value.element[2][0] = 0;
  value.element[2][1] = 0;
  value.element[2][2] = 0;

  return( value );
}

twovector GradA( position zetatau )
{
  twovector value;

  value.element[0][0] = 0;
  value.element[1][0] = 0;
  value.element[2][0] = 0;
  value.element[0][1] = 0;
```

```
  value.element[1][1] = 0;
  value.element[2][1] = 0;

  return( value );
}

vectform LaplacianOmega( position zetatau, specpos Q )
{
  vectform value;

  value.vector[0] = 0;
  value.vector[1] = 0;
  value.vector[2] = 0;

  return( value );
}

double LapdivOmega( position zetatau, specpos Q )
{
  double value;

  value = 0;

  return( value );
}

vectform GraddivOmega( position zetatau, specpos Q )
{
  vectform value;

  value.vector[0] = 0;
  value.vector[1] = 0;
  value.vector[2] = 0;

  return( value );
}
```

```
/*
  qsefuncs.c  --  Collection of functions for QSE paper.  Includes
                  definitions for g, w, Omega.
                                                                  */

#include "flat.h"


/* g is the Green's function defined in (3.3e) */
double g( double kse1, double kse2 )
{
  /* Returns larger * ( 1 - smaller ) */

  if( kse1 > kse2 ){ return( (kse2 * (1-kse1)) ); }
  else{ return( (kse1 * (1-kse2)) ); };
}


/* w is the linear path connecting (y,0) to (x,t) defined in (1.8) */
position w( double kse, specpos Q )
{
  position value;
  int index;

  /* Returns two values:  the vector  y + kse( x-y )
                          the scalar  0 + kse( t-0 ) */

  for( index = 0; index < d; index++ )
    value.vector[index] = Q.y[index] + kse * ( Q.x[index] - Q.y[index] );

  value.scalar = kse * Q.t;

  return( value );
}


/* Omega is defined above (4.32) */
vectform Omega( position zetatau, specpos Q )
{
  vectform value, f_vals;
  matrixform A_vals;
  double difference[d];
  int index, j;

  /* Returns a vector */
```

```
  /* Initialize value to 0 */
  for( index = 0; index < d; index++ )
    value.vector[index] = 0;

  /* Find the value of f at zetatau */
  f_vals = f( zetatau );

  /* Find the value of A at zetatau */
  A_vals = MatrixA( zetatau );

  /* Calculate the vector (x-y)/t */
  for( index = 0; index < d; index++ )
    difference[index] = (Q.x[index] - Q.y[index]) / Q.t;


  /* Combine these values to generate the vector Omega returns
     according to formula                                    */

  for( index = 0; index < d; index++ )
    for( j = 0; j < d; j++ )
        value.vector[index] += difference[j]*A_vals.element[index][j];

  for( index = 0; index < d; index++ )
    value.vector[index] += f_vals.vector[index];

  return( value );
}


/* max_of is used in the calculation of G12 */
double max_of( double kse1, double kse2 )
{
  if( kse1 > kse2 ) return( kse1 );
  return( kse2 );
}


/* d1g is used in the calculation of G02.  It calculates the derivative
     of the Green's function g with respect to the first argument
     passed to g                                             */
double d1g( double kse2, double kse3 )
{
  if( kse2 > kse3 ) return( -kse3 );
  return( 1-kse3 );
}
```

```
/*
  qseinput.c  --  File containing functions defining a & v from the
                  QSE paper.  File is meant to be modified for each
                  new case of a and v.

                  Note:  To change the number of dimensions, change
                  the file flat.h appropriately.

                  Note:  To change the initial points and time,
                  i.e. x, y, t, change the file containing main().

                  Note:  If explicit versions of f, A, divf,
                  partial derivatives of A, etc. are available, modify
                  qsederiv.c appropriately.  If not, link with
                  qsenumer.c instead.


                                                                  */

#include "flat.h"
#include <math.h>

/* v is introduced in (2.5b) */
double v( position XYZt )
{
  /*  XYZt.vector[i] refers to the (i+1)th direction
      e.g.  XYZt.vector[0] = x direction
            XYZt.vector[1] = y direction  in 3 dimensional space
            XYZt.vector[2] = z direction                    */

  /* User defined function */
  double value;

  value = XYZt.vector[0] * XYZt.vector[0] ;

  return( value );
}

/* a is introduced in (2.5a) */
vectform a( position XYZt )
{
  /*  XYZt.vector[i] refers to the (i+1)th direction
      e.g.  XYZt.vector[0] = x direction
            XYZt.vector[1] = y direction  in 3 dimensional space
            XYZt.vector[2] = z direction                    */
```

```
  /* User defined function */
  vectform value;

  value.vector[0] = 0;
  value.vector[1] = 0;
  value.vector[2] = 0;

  return( value );
}
```

```
/*
  qsenumer.c  --  Numerical, non-explicit differentiation of some of
                  the necessary QSE paper's functions.  Use this file
                  when explicit expressions for f, A, divf, and
                  partial derivatives of A are not available.  When
                  they are available, do not link with this file -
                  use the file containing the explicit definitions
                  of those functions.
                                                                  */

#include "flat.h"


/* f is defined in (4.18a) */
vectform f( position zetatau )
{
  /* Numerical differentiation implementation; non-explicit.  Uses
     five point formula for differentiation */

  vectform value;
  position point[4];
  vectform a_vals[4];
  int index, j;

  /* Initialize the points to use in the differentiation formula */
  for( index = 0; index < 4; index++ )
  {
    for( j = 0; j < d; j++ )
        point[index].vector[j] = zetatau.vector[j];
    point[index].scalar = zetatau.scalar;
  }

  for( index = 0; index < d; index++ )
  {
    /* Generate the points to use in the five point numerical
        differentiation formula                          */
    point[0].vector[index] -= 2*h;
    point[1].vector[index] -= h;
    point[2].vector[index] += h;
    point[3].vector[index] += 2*h;

    /* Find the differentials of v with respect to each dimension */
    value.vector[index] = -( v(point[0]) - 8*v(point[1]) + 8*v(point[2])
                                         - v(point[3]) ) / (12*h);
```

```
    /* Restore the points to be equal to zetatau */
    for( j = 0; j < 4; j++ )point[j].vector[index] = zetatau.vector[index];
  }

  /* Generate the points to use in the five point numerical
    differentiation formula to differentiate a wrt t        */
  point[0].scalar -= 2*h;
  point[1].scalar -= h;
  point[2].scalar += h;
  point[3].scalar += 2*h;

  /* Find the values of the function a at those points */
  for( j = 0; j < 4; j++ )a_vals[j] = a(point[j]);

  /* Sum the values to find the differential */
  for( j = 0; j < d; j++ )
    value.vector[j] -= (a_vals[0].vector[j] - 8*a_vals[1].vector[j]
                        + 8*a_vals[2].vector[j]
                        - a_vals[3].vector[j]) / (12*h);

  return( value );
}


/* A is defined in (4.18b) */
matrixform MatrixA( position zetatau )
{
  matrixform value;
  position point[4];
  int i, j, index;
  vectform a_vals[4], a_diffvals[d];

  /* Initialize the points to use in the differentiation formula */
  for( index = 0; index < 4; index++ )
  {
    for( j = 0; j < d; j++ )
        point[index].vector[j] = zetatau.vector[j];
    point[index].scalar = zetatau.scalar;
  }

  for( index = 0; index < d; index++ )
  {
    /* Generate the points to use in the five point numerical
        differentiation formula                          */
    point[0].vector[index] -= 2*h;
```

```
    point[1].vector[index] -= h;
    point[2].vector[index] += h;
    point[3].vector[index] += 2*h;

    /* Find the values of the function a at those points */
    for( j = 0; j < 4; j++ )a_vals[j] = a(point[j]);

    /* Find the differentials */
    for( j = 0; j < d; j++ )
        a_diffvals[index].vector[j] = ( a_vals[0].vector[j] -
                                        8 * a_vals[1].vector[j] +
                                        8 * a_vals[2].vector[j] -
                                        a_vals[3].vector[j] ) / (12*h);

    /* Restore the points to be equal to zetatau */
    for( j = 0; j < 4; j++ )point[j].vector[index] = zetatau.vector[index];
  }


  /* Calculate the matrix A from the differentials */
  for( i = 0; i < d; i++ )
    for( j = 0; j < d; j++ )
    {
        /* Make certain the diagonals are 0 */
        if( i == j ){ value.element[i][j] = 0; }

        else
        {
          value.element[i][j] = a_diffvals[i].vector[j] -
                                a_diffvals[j].vector[i];
        }
    }

  return( value );
}


/* divf will be used in (4.32) */
double divf( position zetatau )
{
  double value;
  position point[4];
  vectform f_vals[4];
  int index, j;
```

```
  value = 0;

  /* Initialize the points to use in the differentiation formula */
  for( index = 0; index < 4; index++ )
  {
    for( j = 0; j < d; j++ )
        point[index].vector[j] = zetatau.vector[j];
    point[index].scalar = zetatau.scalar;
  }

  for( index = 0; index < d; index++ )
  {

    /* Generate the points to use in the five point numerical
        differentiation formula                              */
    point[0].vector[index] -= 2*h;
    point[1].vector[index] -= h;
    point[2].vector[index] += h;
    point[3].vector[index] += 2*h;

    /* Find the values of the function f at those points */
    for( j = 0; j < 4; j++ )f_vals[j] = f(point[j]);

    /* Sum the differentials as a result of the dot product */
    value += (f_vals[0].vector[index] - 8*f_vals[1].vector[index] +
                8*f_vals[2].vector[index] - f_vals[3].vector[index])/(12*h);

    /* Restore the points to equal to zetatau */
    for( j = 0; j < 4; j++ )point[j].vector[index] = zetatau.vector[index];
  }

  return( value );
}


/* GradOmega calcuates differentials of Omega.  Grad wrt i of Omega[j] is
    stored in ".element[j][i]".  This format mimicks that used in
    Mathematica code "Grad[ Omega[[j]] ][[i]]".                    */
matrixform GradOmega( position zetatau, specpos Q )
{
  matrixform value;
  int i, j;
  position point[4];
  vectform Omega_vals[4];
```

```
  /* Initialize the points to use in the differentiation formula */
  for( i = 0; i < 4; i++ )
  {
    for( j = 0; j < d; j++ )
        point[i].vector[j] = zetatau.vector[j];
    point[i].scalar = zetatau.scalar;
  }

  for( i = 0; i < d; i++ )
  {

    /* Generate the points to use in the five point numerical
        differentiation formula                          */
    point[0].vector[i] -= 2*h;
    point[1].vector[i] -= h;
    point[2].vector[i] += h;
    point[3].vector[i] += 2*h;

    /* Find the values of Omega at the 4 points */
    for( j = 0; j < 4; j++ ) Omega_vals[j] = Omega( point[j], Q );

    /* Calculate the differential according to the five point formula */
    for( j = 0; j < d; j++ )
        value.element[j][i] = ( Omega_vals[0].vector[j] -
                                8 * Omega_vals[1].vector[j] +
                                8 * Omega_vals[2].vector[j] -
                                Omega_vals[3].vector[j] ) / (12*h);

    /* Restore the points to equal to zetatau */
    for( j = 0; j < 4; j++ ) point[j].vector[i] = zetatau.vector[i];

  }

  return( value );
}


/* GradA calculates differentials of MatrixA.  Grad wrt i of MatrixA[i][j]
  is calculated, then the implied summation on i is performed to result
  in element[j][0].  Results of Grad wrt i of MatrixA[j][i] are stored in
  element[j][1].  Thus two vectors are returned in the form of a d by 2
  matrix.                                                */
twovector GradA( position zetatau )
{
  twovector value;
```

```
double diffval;
int i, j;
position point[4];
matrixform A_vals[4];

/* Initialize the value to be returned to 0 */
for( i = 0; i < d; i++ )
{
  value.element[i][0] = 0;
  value.element[i][1] = 0;
}

/* Initialize the points to use in the differentiation formula */
for( i = 0; i < 4; i++ )
{
  for( j = 0; j < d; j++ )
       point[i].vector[j] = zetatau.vector[j];
  point[i].scalar = zetatau.scalar;
}

/* Begin differentiation with respect to i */
for( i = 0; i < d; i++ )
{
  /* Generate the points to use in the five point numerical
       differentiation formula                              */
  point[0].vector[i] -= 2*h;
  point[1].vector[i] -= h;
  point[2].vector[i] += h;
  point[3].vector[i] += 2*h;

  /* Find the values of A at the 4 points */
  for( j = 0; j < 4; j++ ) A_vals[j] = MatrixA( point[j] );

  /* Calculate the differential according to the five point formula
    and perform the implied summation on i while keeping track of
    A's symmetry so that unnecessary calculation is avoided.      */
  for( j = 0; j < d; j++ )
  {
       if( i != j )   /* Diagonals of A are always 0   */
       {
         diffval = ( A_vals[0].element[i][j] - 8 * A_vals[1].element[i][j]
                + 8 * A_vals[2].element[i][j] -
                A_vals[3].element[i][j] ) / (12*h);
       value.element[j][0] += diffval;
       value.element[j][1] -= diffval;
```

```
      }
    }

    /* Restore the points to equal to zetatau */
    for( j = 0; j < 4; j++ ) point[j].vector[i] = zetatau.vector[i];

  }

  return( value );
}


/* LaplacianOmega calcuates laplacian of each Omega[j]; used in
    calculation of G12                                    */
vectform LaplacianOmega( position zetatau, specpos Q )
{
  vectform value;
  int i, j;
  position point[5];
  vectform Omega_vals[5];

  /* Initialize the points to use in the differentiation formula */
  for( i = 0; i < 5; i++ )
  {
    for( j = 0; j < d; j++ )
        point[i].vector[j] = zetatau.vector[j];
    point[i].scalar = zetatau.scalar;
  }

  /* Initialize value to 0 */
  for( i = 0; i < d; i++ ) value.vector[i] = 0;

  /* Begin differentiation with respect to i */
  for( i = 0; i < d; i++ )
  {
    /* Generate the points to use in the five point numerical
        differentiation formula                           */
    point[0].vector[i] -= 2*h;
    point[1].vector[i] -= h;

    point[3].vector[i] += h;
    point[4].vector[i] += 2*h;

    /* Find the values of Omega at the 5 points */
    for( j = 0; j < 5; j++ ) Omega_vals[j] = Omega( point[j], Q );
```

```
    /* Calculate the second derivatives individually and sum them
         to get the laplacian according to the five point formula for
         second derivatives                                    */
    for( j = 0; j < d; j++ )
         value.vector[j] += ( -Omega_vals[0].vector[j] +
                              16 * Omega_vals[1].vector[j] -
                              30 * Omega_vals[2].vector[j] +
                              16 * Omega_vals[3].vector[j] -
                              Omega_vals[4].vector[j] ) / (12*h*h);

    /* Restore the points to equal to zetatau */
    for( j = 0; j < 5; j++ ) point[j].vector[i] = zetatau.vector[i];

  }

  return( value );
}


/* DivOmega calculates the divergence of Omega for use in calculating G22 */
double DivOmega( position zetatau, specpos Q )
{
  double value;
  position point[4];
  vectform Omega_vals[4];
  int index, j;

  value = 0;

  /* Initialize the points to use in the differentiation formula */
  for( index = 0; index < 4; index++ )
  {
    for( j = 0; j < d; j++ )
         point[index].vector[j] = zetatau.vector[j];
    point[index].scalar = zetatau.scalar;
  }

  /* Begin differentiation with respect to index */
  for( index = 0; index < d; index++ )
  {
    /* Generate the points to use in the five point numerical
         differentiation formula                               */
    point[0].vector[index] -= 2*h;
    point[1].vector[index] -= h;
```

```
    point[2].vector[index] += h;
    point[3].vector[index] += 2*h;

    /* Find the values of the function Omega at those points */
    for( j = 0; j < 4; j++ )Omega_vals[j] = Omega( point[j], Q );

    /* Sum the differentials as a result of the dot product */
    value += ( Omega_vals[0].vector[index] -
                8 * Omega_vals[1].vector[index] +
                8 * Omega_vals[2].vector[index] -
                Omega_vals[3].vector[index] ) / (12*h);

    /* Restore the points to equal to zetatau */
    for( j = 0; j < 4; j++ )point[j].vector[index] = zetatau.vector[index];
  }

  return( value );
}


/* LapdivOmega calculates the laplacian of the divergence of
   Omega for use in calculating G22                          */
double LapdivOmega( position zetatau, specpos Q )
{
  double value;
  int i, j;
  position point[5];
  double Omega_vals[5];

  /* Initialize the points to use in the differentiation formula */
  for( i = 0; i < 5; i++ )
  {
    for( j = 0; j < d; j++ )
        point[i].vector[j] = zetatau.vector[j];
    point[i].scalar = zetatau.scalar;
  }

  /* Initialize value to 0 */
  value = 0;

  /* Begin differentiation with respect to i */
  for( i = 0; i < d; i++ )
  {
    /* Generate the points to use in the five point numerical
       differentiation formula                                */
```

```
    point[0].vector[i] -= 2*h;
    point[1].vector[i] -= h;

    point[3].vector[i] += h;
    point[4].vector[i] += 2*h;

    /* Find the values of Omega at the 5 points */
    for( j = 0; j < 5; j++ ) Omega_vals[j] = DivOmega( point[j], Q );

    /* Calculate the second derivatives individually and sum them
       to get the laplacian according to the five point formula for
       second derivatives                                    */
    value += ( -Omega_vals[0] + 16 * Omega_vals[1] -
                30 * Omega_vals[2] + 16 * Omega_vals[3] -
                Omega_vals[4] ) / (12*h*h);

    /* Restore the points to equal to zetatau */
    for( j = 0; j < 5; j++ ) point[j].vector[i] = zetatau.vector[i];

  }

  return( value );
}


/* DeliOmegai calculates the partial derivative of Omega[i] wrt i for
    use in calculating G12                                   */
vectform DeliOmegai( position zetatau, specpos Q )
{
  vectform value;
  int i, j;
  position point[4];
  vectform Omega_vals[4];

  /* Initialize the points to use in the differentiation formula */
  for( i = 0; i < 4; i++ )
  {
    for( j = 0; j < d; j++ )
       point[i].vector[j] = zetatau.vector[j];
    point[i].scalar = zetatau.scalar;
  }

  /* Begin differentiation with respect to i */
  for( i = 0; i < d; i++ )
  {
```

```
    /* Generate the points to use in the five point numerical
         differentiation formula                              */
    point[0].vector[i] -= 2*h;
    point[1].vector[i] -= h;
    point[2].vector[i] += h;
    point[3].vector[i] += 2*h;

    /* Find the values of Omega at the 4 points */
    for( j = 0; j < 4; j++ ) Omega_vals[j] = Omega( point[j], Q );

    /* Calculate the differential according to the five point formula */
    value.vector[i] = ( Omega_vals[0].vector[i] -
                        8 * Omega_vals[1].vector[i] +
                        8 * Omega_vals[2].vector[i] -
                        Omega_vals[3].vector[i] ) / (12*h);

    /* Restore the points to equal to zetatau */
    for( j = 0; j < 4; j++ ) point[j].vector[i] = zetatau.vector[i];

  }

  return( value );
}


/* GraddivOmega calculates the partial second derivative of Omega[i]
     wrt i wrt j, calculates the implied summation on i into
     vector[j], thus returning a vector.  For use in calcuating G12. */
vectform GraddivOmega( position zetatau, specpos Q )
{
  vectform value;
  int i, j;
  position point[4];
  vectform Omega_vals[4];

  /* Initialize the points to use in the differentiation formula */
  for( i = 0; i < 4; i++ )
  {
    for( j = 0; j < d; j++ )
         point[i].vector[j] = zetatau.vector[j];
    point[i].scalar = zetatau.scalar;
  }

  /* Initialize the value to be returned */
  for( i = 0; i < d; i++ )
```

```
    value.vector[i] = 0;

  /* Begin differentiation with respect to i */
  for( i = 0; i < d; i++ )
  {
    /* Generate the points to use in the five point numerical
         differentiation formula                              */
    point[0].vector[i] -= 2*h;
    point[1].vector[i] -= h;
    point[2].vector[i] += h;
    point[3].vector[i] += 2*h;

    /* Find the values of DeliOmegai at the 4 points */
    for( j = 0; j < 4; j++ ) Omega_vals[j] = DeliOmegai( point[j], Q );

    /* Calculate the differential according to the five point formula */
    for( j = 0; j < d; j++ )
         value.vector[i] += ( Omega_vals[0].vector[j] -
                              8 * Omega_vals[1].vector[j] +
                              8 * Omega_vals[2].vector[j] -
                              Omega_vals[3].vector[j] ) / (12*h);

    /* Restore the points to equal to zetatau */
    for( j = 0; j < 4; j++ ) point[j].vector[i] = zetatau.vector[i];

  }

  return( value );
}
```

```
/*
   qsetj.c  --  Contains functions necessary for the calculation of
                T1, T2, and J from the QSE paper.
                                                                    */

#include "flat.h"
#define SIMPSON 1
#define ROMBERG 2

/* dG01 is the function integrated to calculate G01 and then T1 */
double dG01( double kse1, double kse2, specpos Q )
{
   double value;
   vectform temp1, temp2;
   int index;

   value = 0;

   temp1 = Omega( w( kse1, Q ), Q );
   temp2 = Omega( w( kse2, Q ), Q );

   for( index = 0; index < d; index++ )
      value += temp1.vector[index] * temp2.vector[index];
   value = value * g( kse1, kse2 );

   return( value );
};


/* dG11 is the function integrated to calculate G11 and then T1 */
double dG11( double kse1, specpos Q )
{
   double value;
   int j;
   twovector DelA_vals;

   value = 0;

   DelA_vals = GradA( w(kse1,Q) );

   for( j = 0; j < d; j++ )
      value += ( Q.x[j] - Q.y[j] ) * DelA_vals.element[j][0];

   value = (kse1 - kse1*kse1) * ( divf( w(kse1,Q) ) + value/Q.t );
```

```
  return( value );
};


/* T1 is defined in (4.32) */
T1form T1( specpos Q, int numofpts )
{
  T1form value;

  /* T1 returns two scalar values.  To combine, multiply the first
     value by 1/(i*hbar) and add to the second value.          */

  /* Calculate G^1_0 */
  value.ih_neg1 = (.5) * Q.t * Q.t * Q.t *
                  Simp_DblIntegrate( dG01, 0,1, 0,1, numofpts, Q );

  /* Calculate G^1_1 */
  value.ih_zero = -(.5) * Q.t * Q.t *
                  Simp_Integrate( dG11, 0,1, numofpts, Q );

  return( value );
}


/* J1 is exactly equal to T1 */
T1form J1( specpos Q, int numofpts )
{
  T1form value;

  value = T1( Q, numofpts );

  return( value );
}


/* dG02 calculates the function to be integrated to calculate G02 and
    then T2;  G02 is defined in (4.33b)                    */
double dG02( double kse1, double kse2, double kse3, specpos Q )
{
  double value;
  int i, j;
  vectform Omega1_vals, Omega3_vals;
  matrixform DelOmega2_vals;
  matrixform A2_vals;
```

```
  value = 0;
  Omega1_vals = Omega( w(kse1,Q), Q );
  Omega3_vals = Omega( w(kse3,Q), Q );
  DelOmega2_vals = GradOmega( w(kse2,Q), Q );
  A2_vals = MatrixA( w(kse2,Q) );

  for( i = 0; i < d; i++ )
    for( j = 0; j < d; j++ )
      value += Omega1_vals.vector[i] * Omega3_vals.vector[j] *
            ( g(kse2,kse3) * DelOmega2_vals.element[i][j] +
              d1g(kse2,kse3) / Q.t * A2_vals.element[i][j] );

  value = value * g(kse1,kse2);

  return( value );
}


/* dG12 calculates the function to be integrated to calculate G12 and
    then T2;  G12 is defined in (4.33c)                          */
double dG12( double kse1, double kse2, specpos Q )
{
  double part1, part2, part3, value;
  matrixform DelOmega1_vals, DelOmega2_vals;
  matrixform A1_vals, A2_vals;
  vectform Omega1_vals;
  vectform LapOmega2_vals;
  twovector DelA2_vals;
  vectform DblDelOmega2_vals;
  int i, j;

  value = 0;

  if( kse1 == 0 )kse1 = (double)1e-20;
  if( kse2 == 0 )kse2 = (double)1e-20;

  part1 = 0;
  DelOmega1_vals = GradOmega( w(kse1,Q), Q );
  DelOmega2_vals = GradOmega( w(kse2,Q), Q );
  A1_vals = MatrixA( w(kse1,Q) );
  A2_vals = MatrixA( w(kse2,Q) );

  for( i = 0; i < d; i++ )
    for( j = 0; j < d; j++ )
      part1 += ( DelOmega1_vals.element[j][i] +
```

```
                A1_vals.element[j][i] / (kse1*Q.t) ) *
              ( DelOmega2_vals.element[j][i] +
                A2_vals.element[j][i] / (kse2*Q.t) );
  part1 = part1 * g(kse1,kse2);


  part2 = 0;
  Omega1_vals = Omega( w(kse1,Q), Q );
  LapOmega2_vals = LaplacianOmega( w(kse2,Q), Q );
  DelA2_vals = GradA( w(kse2,Q) );

  for( j = 0; j < d; j++ )
    part2 += Omega1_vals.vector[j] * ( LapOmega2_vals.vector[j] +
          2 / (kse2*Q.t) * DelA2_vals.element[j][1] );
  part2 = part2 * kse2 * kse2 * ( 1/max_of(kse1,kse2) - 1 );


  part3 = 0;
  DblDelOmega2_vals = GraddivOmega( w(kse2,Q), Q );

  for( j = 0; j < d; j++ )
    part3 += Omega1_vals.vector[j] * ( DblDelOmega2_vals.vector[j] +
          DelA2_vals.element[j][0] / (kse2*Q.t) );
  part3 = part3 * kse2 * ( 2 - ( 1 + 1/max_of(kse1,kse2) )*kse2 );


  value = g(kse1,kse2) * (part1 + part2 + part3);

  return( value );
}


/* dG22 calculates the function to be integrated to calculate G22 and
    then T2;  G22 is defined in (4.33d)                          */
double dG22( double kse1, specpos Q )
{
  double value;

  value = ( kse1 * (1-kse1) ) * ( kse1 * (1-kse1) ) *
        LapdivOmega( w(kse1,Q), Q );

  return( value );
}
```

```
/* T2 is defined in (4.33a) */
T2form T2( specpos Q, int numofpts, int tplintgscheme )
{
  T2form value;
  T1form T1value;

  T1value = T1( Q, numofpts );

  value.ih_neg2 = .5 * T1value.ih_neg1 * T1value.ih_neg1;
  value.ih_neg1 = T1value.ih_neg1 * T1value.ih_zero;
  value.ih_zero = .5 * T1value.ih_zero * T1value.ih_zero;

  if( tplintgscheme == ROMBERG )
  {
    value.ih_neg1 += -.5 * Q.t*Q.t*Q.t*Q.t*Q.t *
                Rmbg_Simplex_TplIntegrate( dG02, 0,1, 0,1, 0,1,
                                numofpts, Q );
  }
  else
  {
    value.ih_neg1 += -.5 * Q.t*Q.t*Q.t*Q.t*Q.t *
                Simp_Simplex_TplIntegrate( dG02, 0,1, 0,1, 0,1,
                                numofpts, Q );
  }

  value.ih_zero += .25 * Q.t*Q.t*Q.t*Q.t *
              Simp_DblIntegrate( dG12, 0,1, 0,1, numofpts, Q );

  value.ih_pos1 = -.125 * Q.t*Q.t*Q.t *
             Simp_Integrate( dG22, 0,1, numofpts, Q );

  return( value );
}


/* J2 is equal to T2 - (1/2) T1^2 */
T2form J2( specpos Q, int numofpts, int tplintgscheme )
{
  T2form value;

  if( tplintgscheme == ROMBERG )
  {
    value.ih_neg1 = -.5 * Q.t*Q.t*Q.t*Q.t*Q.t *
                Rmbg_Simplex_TplIntegrate( dG02, 0,1, 0,1, 0,1,
                                numofpts, Q );
```

```
    }
    else
    {
      value.ih_neg1 = -.5 * Q.t*Q.t*Q.t*Q.t*Q.t *
                  Simp_Simplex_TplIntegrate( dG02, 0,1, 0,1, 0,1,
                                    numofpts, Q );
    }

    value.ih_zero = .25 * Q.t*Q.t*Q.t*Q.t *
                Simp_DblIntegrate( dG12, 0,1, 0,1, numofpts, Q );

    value.ih_pos1 = -.125 * Q.t*Q.t*Q.t *
                Simp_Integrate( dG22, 0,1, numofpts, Q );

    return( value );
}


/* dJ calculates the function to be integrated to calculate phase factor J */
double dJ( double kse, specpos Q )
{
    double value;
    int index;
    vectform a_val;
    position zetatau;

    zetatau = w( kse, Q );

    a_val = a( zetatau );

    value = Q.t * v( zetatau );

    for( index = 0; index < d; index++ )
      value -= (Q.x[index] - Q.y[index]) * a_val.vector[index];

    return( value );
}

/* J calculates the phase factor J defined in (1.7) */
double J( specpos Q, int numofpts )
{
    double value;

    value = Simp_Integrate( dJ, 0,1, numofpts, Q );
    return( value );
}
```

```
/*
  tjmain.c  --  Contains main for calculation of J, T1, T2

                Note:  To change the number of dimensions, modify
                the file flat.h appropriately.

                Note:  To change a and v, modify the file qseinput.c
                appropriately.

                Note:  To change initial values x, y and t, modify
                the variable Q.  Q.x refers to vector x, Q.y refers
                to vector y, Q.t refers to time t.  Vectors x and
                y are stored as arrays.  e.g. Q.x[0] = 2.5;
                                             Q.t = 3;

                Note:  If explicit versions of f, A, divf,
                partial derivatives of A, etc. are available, modify
                qsederiv.c appropriately.  If not, link with
                qsenumer.c instead.

                                                                */

#include <stdio.h>
#include "flat.h"
#define SIMPSON 1
#define ROMBERG 2

int main( )
{
  T2form result;
  T1form t1result;
  specpos Q;
  int numofpts;
  int i;
  double value;
  double jresult;

  /* Set value for y, t and let x vary, calculate J2 */
/*
  printf( "\nJ2 with x vary from -5 to 5 with t=3.5 and y[0]=2" );
*/
  Q.y[0] = 2;
  Q.y[1] = 0;
  Q.y[2] = 0;
```

```
  Q.t = (double)3.5;

  Q.x[1] = 0;
  Q.x[2] = 0;

  numofpts = 20;

  for( i = 0, Q.x[0] = -5; i < 200; i++, Q.x[0] += (double).05 )
  {
      result = J2( Q, numofpts, ROMBERG );
      printf( "\n%e %e %e", result.ih_neg1,
                        result.ih_zero, result.ih_pos1 );
  }


  /* Set value for y, t and let x vary, calculate T1 */
/*
  printf( "\nT1 with x vary from -5 to 5 with t=3.5 and y[0]=2" );

  Q.y[0] = 2;
  Q.y[1] = 0;
  Q.y[2] = 0;

  Q.t = (double)3.5;

  Q.x[1] = 0;
  Q.x[2] = 0;

  numofpts = 20;

  for( i = 0, Q.x[0] = -5; i < 200; i++, Q.x[0] += (double).05 )
  {
    t1result = T1( Q, numofpts );
    printf( "\n%e %e", t1result.ih_neg1, t1result.ih_zero );
  }
*/
  /* Set value for y, t and let x vary, calculate J */
/*
  printf( "\nJ with x vary from -5 to 5 with t=3.5 and y[0]=2" );

  Q.y[0] = 2;
  Q.y[1] = 0;
  Q.y[2] = 0;

  Q.t = 3.5;
```

```
  Q.x[1] = 0;
  Q.x[2] = 0;

  numofpts = 20;

  for( i = 0, Q.x[0] = -5; i < 200; i++, Q.x[0] += (double).05 )
  {
    jresult = J( Q, numofpts );
    printf( "\n%e", jresult );
  }
*/

  return( 1 );
}
```

Accompanying Mathematica Source Code

The following four pages contain Mathematica source code that was developed in tandem with the C code preceding it. Its sole purpose was to provide a check for programming errors in the C code. This Mathematica code is limited to 3-dimensional coordinate space. Its other major limitation is its inability to evaluate some of the integrals in the calculation of T_1 and T_2. Note that the C code is capable of working in n-dimensional space.

Mathematica source code

```
<<Calculus`VectorAnalysis`

SetCoordinates[ Cartesian[ x, y, z ] ]

Omega[ xd_, yd_, zd_, td_ ] := Block[ {x,y,z,t,fvals,Avals,value},
   fvals = f[x,y,z,t]; Avals = A[x,y,z,t]; value =
   { fvals[[1]]+
((x1-y1)*Avals[[1]][[1]]+(x2-y2)*Avals[[1]][[2]]+(x3-y3)*Avals[[1]][[3]])/t,
    fvals[[2]]+
((x1-y1)*Avals[[2]][[1]]+(x2-y2)*Avals[[2]][[2]]+(x3-y3)*Avals[[2]][[3]])/t,
    fvals[[3]]+
((x1-y1)*Avals[[3]][[1]]+(x2-y2)*Avals[[3]][[2]]+(x3-y3)*Avals[[3]][[3]])/t};
   x=xd; y=yd; z=zd; t=td; value ]


A[ xd_, yd_, zd_, td_ ] := Block[ {x,y,z,t,value}, value =
   {{0, D[a[x,y,z,t][[2]], x] - D[a[x,y,z,t][[1]], y],
      D[a[x,y,z,t][[3]], x] - D[a[x,y,z,t][[1]], z]},
    {D[a[x,y,z,t][[1]], y] - D[a[x,y,z,t][[2]], x], 0,
     D[a[x,y,z,t][[3]], y] - D[a[x,y,z,t][[2]], z]},
    {D[a[x,y,z,t][[1]], z] - D[a[x,y,z,t][[3]], x],
     D[a[x,y,z,t][[2]], z] - D[a[x,y,z,t][[3]], y], 0}};
    x=xd; y=yd; z=zd; t=td; value ]

deldotf[ xd_, yd_, zd_, td_ ] := Block[ {x,y,z,t,value}, value =
       D[f[x,y,z,t][[1]], x] + D[f[x,y,z,t][[2]], y] +
       D[f[x,y,z,t][[3]], z];  x=xd; y=yd; z=zd; t=td; value ]

specialT1[ xd_, yd_, zd_, td_ ] := Block[ {x,y,z,t,value,Avals},
   Avals = A[x,y,z,t]; value =
   (x1-y1)/t * D[Avals[[1]][[1]],x] + (x2-y2)/t * D[Avals[[1]][[2]],x] +
                        (x3-y3)/t * D[Avals[[1]][[3]],x] +
   (x1-y1)/t * D[Avals[[2]][[1]],y] + (x2-y2)/t * D[Avals[[2]][[2]],y] +
                        (x3-y3)/t * D[Avals[[2]][[3]],y] +
   (x1-y1)/t * D[Avals[[3]][[1]],z] + (x2-y2)/t * D[Avals[[3]][[2]],z] +
                        (x3-y3)/t * D[Avals[[3]][[3]],z];
   x=xd; y=yd; z=zd; t=td; value ]


f[ xd_, yd_, zd_, td_ ] := Block[ {x,y,z,t,value}, value =
   {-D[v[x,y,z,t], x] - D[a[x,y,z,t][[1]], t],
    -D[v[x,y,z,t], y] - D[a[x,y,z,t][[2]], t],
```

```
    -D[v[x,y,z,t], z] - D[a[x,y,z,t][[3]], t]};
    x=xd; y=yd; z=zd; t=td; value ]

v[ x_, y_, z_, t_ ] := d*x + c*x^2

a[ x_, y_, z_, t_ ] := {a1*x*y, a2*y^2 + a3*z^2, a5*z + a4*z^2}

v[ x_, y_, z_, t_ ] := Sin[x]
a[ x_, y_, z_, t_ ] := {0,0,0}

g[kse1_, kse2_] := Min[kse1, kse2]*(1 - Max[kse1, kse2])

w[kse_] := {y1 + kse*(x1 - y1), y2 + kse*(x2 - y2), y3 + kse*(x3 - y3),
  kse*tt}

v[wl_] := v[ wl[[1]], wl[[2]], wl[[3]], wl[[4]] ]
a[wl_] := a[ wl[[1]], wl[[2]], wl[[3]], wl[[4]] ]

f[wl_] := f[ wl[[1]], wl[[2]], wl[[3]], wl[[4]] ]

deldotf[wl_] :=
   deldotf[ wl[[1]], wl[[2]], wl[[3]], wl[[4]] ]

A[wl_] := A[ wl[[1]], wl[[2]], wl[[3]], wl[[4]] ]

Omega[ wl_ ] :=
   Omega[ wl[[1]], wl[[2]], wl[[3]], wl[[4]] ]

specialT1[wl_] :=
   specialT1[ wl[[1]], wl[[2]], wl[[3]], wl[[4]] ]

dG01[kse1_,kse2_] := Block[{Omega1val, Omega2val, value},
     Omega1val = Omega[w[kse1]]; Omega2val = Omega[w[kse2]];
     value = 2*kse1*(1 - kse2)*
          (Omega1val[[1]]*Omega2val[[1]] +
           Omega1val[[2]]*Omega2val[[2]] +
           Omega1val[[3]]*Omega2val[[3]]);
     value]

dG11[kse1_] := (kse1-kse1^2)*(deldotf[ w[kse1] ] + specialT1[ w[kse1] ])

T1[yd1_, yd2_, yd3_, xd1_, xd2_, xd3_, tdt_] := Block[ {value},
 y1=yd1; y2=yd2; y3=yd3; x1=xd1; x2=xd2; x3=xd3; tt=tdt;
 value = (tt^3*Integrate[Integrate[dG01[kse1, kse2], {kse1, 0, kse2}],
    {kse2, 0, 1}])/(2*I*h) - (tt^2*Integrate[dG11[kse1], {kse1, 0, 1}])/2;
 y1=yd1; y2=yd2; y3=yd3; x1=xd1; x2=xd2; x3=xd3; tt=tdt;
```

```
  value ]

d1g[ kse2_, kse3_ ] := If[ kse2 > kse3, -kse3, (1-kse3) ]

dG02[ kse1_, kse2_, kse3_ ] := Block[ {value,i,j},
  value = g[kse1,kse2] *
  Sum[ Omega[w[kse1]][[i]] * Omega[w[kse3]][[j]] *
    ( g[kse2,kse3] * Block[{x,y,z,t,temp},
                temp = Grad[Omega[x,y,z,t][[i]]][[j]];
                {x,y,z,t}=w[kse2]; temp ] +
    d1g[kse2,kse3] / tt * A[w[kse2]][[i]][[j]] ), {i,1,3}, {j,1,3} ];
  value ]

dG12[ kse1_, kse2_ ] := Block[ {value,i,j,part1,part2,part3},
  part1 = g[kse1,kse2] * Sum[
    ( Block[{x,y,z,t,temp},
     temp = Grad[Omega[x,y,z,t][[j]]][[i]];
     {x,y,z,t} = w[kse1]; temp ] + A[w[kse1]][[j]][[i]] / (kse1*tt) ) *
    ( Block[{x,y,z,t,temp},
     temp = Grad[Omega[x,y,z,t][[j]]][[i]];
     {x,y,z,t} = w[kse2]; temp ] + A[w[kse2]][[j]][[i]] / (kse2*tt) ),
    {i,1,3}, {j,1,3} ];
  part2 = kse2^2 * ( 1/Max[kse1,kse2] - 1 ) * Sum[
    Omega[w[kse1]][[j]] * ( Block[{x,y,z,t,temp},
                temp = Laplacian[Omega[x,y,z,t][[j]]];
                {x,y,z,t}=w[kse2]; temp ] +
    2 / (kse2*tt) * Block[{x,y,z,t,temp},
              temp = Grad[A[x,y,z,t][[j]][[i]]][[i]];
              {x,y,z,t} = w[kse2]; temp ] ),
    {i,1,3}, {j,1,3} ];
  part3 = kse2 * ( 2 - kse2 * ( 1 + 1/Max[kse1,kse2] ) ) * Sum[
    Omega[w[kse1]][[j]] * ( Block[{x,y,z,t,temp},
                temp = Grad[Grad[Omega[x,y,z,t][[i]]][[i]]][[j]];
                {x,y,z,t} = w[kse2]; temp ] +
    1 / (kse2*tt) * Block[{x,y,z,t,temp},
              temp = Grad[A[x,y,z,t][[i]][[j]]][[i]];
              {x,y,z,t} = w[kse2]; temp ] ),
    {i,1,3}, {j,1,3} ];
  value = g[kse1,kse2] * ( part1 + part2 + part3 );
  value ]

dG22[ kse1_ ] := Block[ {value}, value = (kse1*(1-kse1))^2;
  Block[ {x,y,z,t,temp}, temp = Laplacian[Div[Omega[x,y,z,t]]];
    {x,y,z,t} = w[kse1]; value = value * temp ];
  value ]
```

```
T2[ yd1_, yd2_, yd3_, xd1_, xd2_, xd3_, tdt_, n_ ] := Block[ {value},
  y1=yd1; y2=yd2; y3=yd3; x1=xd1; x2=xd2; x3=xd3; tt=tdt;
  value = .5 * T1[ yd1,yd2,yd3, xd1,xd2,xd3, tdt ]^2 +
  (I h)^-1 * -.5 * tt^5 * N[ Integrate[ dG02[kse1,kse2,kse3],
                    {kse1,0,1}, {kse2,0,1}, {kse3,0,1} ], n ] +
  .25 * tt^4 * N[ Integrate[ dG12[kse1,kse2], {kse1,0,1}, {kse2,0,1} ], n ] +
  (I h) * -.125 * tt^3 * N[ Integrate[ dG22[kse1], {kse1,0,1} ], n ];
  y1=yd1; y2=yd2; y3=yd3; x1=xd1; x2=xd2; x3=xd3; tt=tdt;
  Expand[value] ]

J[ yd1_, yd2_, yd3_, xd1_, xd2_, xd3_, tdt_ ] := Block[ {xvect,yvect,value},
  y1=yd1; y2=yd2; y3=yd3; x1=xd1; x2=xd2; x3=xd3; tt=tdt;
  xvect = { xd1, xd2, xd3 }; yvect = { yd1, yd2, yd3 };
  value = Integrate[ tt * v[w[kse]] - ( xvect - yvect ) . a[w[kse]],
            {kse,0,1} ];
   y1=yd1; y2=yd2; y3=yd3; x1=xd1; x2=xd2; x3=xd3; tt=tdt;
  value ]
```

Appendix C

Connected Graph Expansion Mathematica Source Code

Description of Mathematica Connected Graph Expansion Routines

All references to "the formula" refer to the main equation appearing in Theorem 1.1 of Fulling and Borosh, *Cataloguing General Graphs by Point and Line Spectra*.

- First, we generate a master list of ξ_i variables.

```
kselist={kse1,kse2,kse3,kse4,kse5,kse6,kse7,kse8,kse9,
         kse10,kse11,kse12,kse13,kse14,kse15,kse16};
```

- Next we define the function `Ilist` to be used in the main routine `L`. `Ilist` generates a list of limits for the various integrals to be evaluated in `L`. These limits result from the expansion of integrals involving green's functions, $g(\xi_i,\xi_j)$, into two separate integrals. This is explained in more depth later on.

```
Ilist[nm_] := Block[ {ILIST,tt}, ILIST={};
              If[nm==0,ILIST={{0,1}},
                 For[ tt=1, tt<=nm, tt++,
                      AppendTo[ILIST,kselist[[tt]]]];
                 ILIST=Permutations[ILIST];
                 For[ tt=1, tt<=nm!, tt++,
                      PrependTo[ILIST[[tt]],0];
                      AppendTo[ILIST[[tt]],1]]];
                 ILIST ]
```

- Now we define the main function, `L`. `L` takes three inputs: `p` is the number of points and `q` is the number of lines for the general graph represented by its adjacency matrix `lgrph`.

```
L[p_,q_,lgrph_] := Block[{pmax,G,tp,tt,alpha,i,j,l,dlist,
                          dpart,dtemp,IList,r,Ipart,n,
                          lfactor,qpart,tr,Mlist},
```

  - `pmax` is equal to the number of elements in the upper triangle of the input adjacency matrix (the nonredundant part of the matrix) of dimension `p`. We store `lgrph` in the new variable `G` and will no longer refer to `lgrph`. (This is because *Mathematica* would object to some manipulations of the input variable `lgrph`.)

```
pmax = 1/2 * p * (p+1);
G = lgrph;
```

- Two lists, `i` and `j`, are created to provide a 1-1 mapping between the nonredundant part of the adjacency matrix and the set of integers from 1 to `pmax`. This mapping is defined as `l[alpha]`.

```
i = {};
For[ tp=1, tp <= p, tp++,
     i = Join[i,Table[ tt, {tt,1,tp} ]] ];
j = {};
For[ tp=1, tp <= p, tp++,
     j = Join[j,Table[ tp, {tt,1,tp} ]] ];
     l[alpha_] := G[[i[[alpha]],j[[alpha]]]];
```

- By limiting ourselves to a 1-dimensional potential function `v`, we greatly simplify the task of keeping track of which derivatives appear in the formula. Del's simplify to a single partial derivative and dot products become simple commutative multiplication (e.g. $\nabla_i \bullet \nabla_j$ becomes $\partial_i \partial_j$.) Thus the list `dlist` is created to keep track of how many times, for each i, a derivative with respect to ξ_i appears in the formula. Here the variable `alpha` is being reused as an index.

```
dlist = Table[ 0, {alpha, 1, p} ];
For[ alpha=1, alpha <= pmax, alpha++,
     dlist[[i[[alpha]]]] += l[alpha];
     dlist[[j[[alpha]]]] += l[alpha] ];
```

- The derivatives only act on the product of potentials, $\prod_{i=1}^{p} v(z_i)$. Now that we know which and how many of each derivative appear, we let the derivatives act on the product of potentials and store the result in `dpart`. Notice that substitutions are made so that `dpart` is expressed in terms of ξ_i's.

```
dpart = 1;
For[ alpha=1, alpha <= p, alpha++,
     dtemp = (D[v[r],{r,dlist[[alpha]]}]);
     dpart *= dtemp /.
               r->(y0-kselist[[alpha]]*(y0-x0))];
```

- What remains are the green's functions. Since we wish to symbolically evaluate the integrals in the formula, we must separate the integrals involving green's functions into two separate integrals:

$$\int_a^b d\xi_i \int_c^d d\xi_j \, \mathrm{F}(\xi_i,\xi_j)\,\mathrm{g}(\xi_i,\xi_j) = \int_a^b d\xi_i \Big(\int_c^{\xi_i} d\xi_j \, \mathrm{F}(\xi_i,\xi_j)\xi_j(\xi_i - 1) + \int_{\xi_i}^d d\xi_j \, \mathrm{F}(\xi_i,\xi_j)\xi_i(\xi_j - 1) \Big)$$

Multiple such separations can result in duplicating integrations. These factors are removed as the final step in `L`. For green's functions of only one variable, i.e. $g(\xi_i,\xi_i) = \xi_i (\xi_i - 1)$, no such separation is warranted. These green's functions of one

variable result from nonzero elements along the diagonal of the adjacency matrix. Thus, we first multiply all factors of $g(\xi_i,\xi_i)$ into `dpart`.

```
For[ alpha=1, alpha <= p, alpha++,
     dpart *= (kselist[[alpha]]*
               (kselist[[alpha]]-1))^G[[alpha,alpha]]];
```

- Now we begin the complicated task of splitting up the integrals by green's functions. Through each loop of this top `For[]` loop we will deal with the integral over the ξ_{tp+1} variable. The result from each level of integration (this top `For[]` loop) is stored in `Ipart`.

```
For[ tp=p-1, tp>=0, tp--,
     Ipart = 0;
```

  - We store the list of limits of the integrals over ξ_{tp-1} in `IList`. We then create a mirror list, `Mlist`, for the sake of convenience, containing the indices of the ξ_i's that appear in `IList`. For example, if `IList`={0, kse2, kse3, kse1, 1} then `Mlist`={2, 3, 1}.

```
IList = Ilist[tp];
Mlist = Permutations[Table[tt,{tt,1,tp}]];
```

  - We will have `tp!` permutations of $\{0, \xi_1, \xi_2, \ldots, \xi_{tp}, 1\}$ contained in the list `IList`. Through each loop of this `For[]` loop we will deal with the `alpha`th permutation. Through each loop of the next `For[]` loop we will be integrating from positions `tt` to `tt+1` in the `alpha`th permutation.

```
For[ alpha=1, alpha<=tp!, alpha++,
     For[ tt=1, tt<=tp+1, tt++,
          gpart = 1;
```

    - Multiply in each factor of $g(\xi_{tr-1},\xi_{tp-1})$.

```
          For[ tr=1, tr<=tp, tr++,
               If[ tr < tt, gpart *=
                    (IList[[alpha]][[tr+1]]*
                    (kselist[[tp+1]]-1)
                    )^G[[tp+1,Mlist[[alpha]][[tr]]]],
                    gpart *= (kselist[[tp+1]]*
                    (IList[[alpha]][[tr+1]]-1)
                    )^G[[tp+1,Mlist[[alpha]][[tr]]]]
                 ]
             ];
```

    - Now integrate over ξ_{tp-1}.

```
          Ipart += Integrate[ gpart*dpart,
                    {kselist[[tp+1]],
                    IList[[alpha]][[tt]],
                    IList[[alpha]][[tt+1]]} ]
        ]
  ];
```

```
    dpart = Ipart
  ];
```

- Calculate and factor in the necessary prefactors from the formula.

```
n = Sum[G[[ta,ta]],{ta,1,p}];
lfactor=Product[l[alpha]!,{alpha,1,pmax}];
Ipart = Ipart / (p! lfactor 2^n);
```

- Calculate and factor out the duplicating integrations.

```
Ipart = Ipart / (Product[ta!,{ta,1,p-1}]);
```

- Return the result.

```
Ipart ]
```

```
LExpSeries[condfn_] :=
      Block[{p,q,lgrph,maxgrph,G,pmax,tp,tr,tt,alpha,
             i,j,l,dlist,dpart,dtemp,iresult,n,lfactor,
             r,A,B,sumovergraphs},
      Clear[hbar,t,m,lambda];
      q=0;
      iresult = 0;
      A = (hbar*t)/m;
      B = (lambda*t)/hbar;
      For[p=1, condfn[p,q]!=0, p++,
         pmax = 1/2 * p * (p+1);
         For[q=0, condfn[p,q]!=0, q++,
            lgrph = labGraphs[p,q];
            maxgrph = Length[lgrph];
            sumovergraphs = 0;
            For[tt=1, tt<=maxgrph, tt++,
               G=lgrph[[tt,1]];
               If[ ConnGraphQ[G,p,q][[3]],
                  i = {};
                  For[ tp=1, tp <= p, tp++,
                       i = Join[i,Table[ tt, {tt,1,tp} ]] ];
                  j = {};
                  For[ tp=1, tp <= p, tp++,
                       j = Join[j,Table[ tp, {tt,1,tp} ]] ];
                  l[alpha_] := G[[i[[alpha]],j[[alpha]]]];

                  dlist = Table[ 0, {tt, 1, p} ];
                  For[ alpha=1, alpha <= pmax, alpha++,
                       dlist[[i[[alpha]]]] += l[alpha];
                       dlist[[j[[alpha]]]] += l[alpha] ];

                  dpart = 1;
                  For[ alpha=1, alpha <= p, alpha++,
                       dtemp = (D[v[r],
                                {r,dlist[[alpha]]}]);
                       dpart *= dtemp /.
                             r->(y0-kselist[[alpha]]*(y0-x0))];

                  For[ alpha=1, alpha <= pmax, alpha++,
                       dpart *= (kselist[[i[[alpha]]]]*
                                 (kselist[[j[[alpha]]]]-1)
                                 )^l[alpha]];

                  Off[Power::infy];
                  For[ tr=p, tr > 1, tr--,
                     dpart = Integrate[ dpart, {kselist[[tr]],
                                        0,kselist[[tr-1]]} ] ];
                  dpart = Integrate[ dpart,{kselist[[1]],0,1}];
                  sumovergraphs += dpart;
                  On[Power::infy]
                  ]
```

```
            ];
          iresult += (-I)^(p + q)*A^q*B^p*sumovergraphs ];
        q=0 ];

        n = Sum[G[[tp,tp]],{tp,1,p}];
        lfactor=Product[l[alpha]!,{alpha,1,pmax}];
        iresult = iresult / (p! lfactor 2^n);
        iresult
    ]
ConnGraphQ[G_,p_,q_] := Block[{qleft,index,value},
   (* The following algorithm arbitrarily begins at
      point 1.  As it decides which points in the
      graph are connected to point 1, it uses a single
      data structure to store necessary information.
      If the algorithm finds that all points in the
      graph are connected to point 1, then the graph
      IS connected.  The data structure has the
      format: {# lines available to connect new
      points into the following list of points, {list
      of points connected to point 1 so far}}.  The
      variable "value" contains this data.              *)
   qleft = q - Sum[G[[index,index]],{index,1,p}];
   value = { qleft, { 1 } };

   (* Begin "walking" across the graph, starting
      at position (1,1) on the adjacency matrix G.
      Store the compiled data back into "value".    *)
   value = Gwalk[ G, p, q, 1, 1, 1, 0, value ];

   (* If all points in the graph are connected to
      point 1, then the graph is connected (True).
      Append the Boolean values True or False to
      "value" and output all of "value".            *)
   If[ Length[value[[2]]] != p, AppendTo[ value, False ],
                                AppendTo[ value, True ]];
   value ]
Gwalk[G_,p_,q_,initi_,initj_,inci_,incj_,value_] :=
   (* This algorithm recursively "walks" across a
      graph according to which points are connected
      to point 1.  It is designed for use with
      ConnGraphQ.  See ConnGraphQ for background
      information.                                   *)
   (* (initi,initj) is the point on the adjacency
      matrix the algorithm will walk FROM.  inci &
      incj describe the direction to walk.          *)
   Block[ {newvalue,k}, newvalue = value;
   (* If all points are already connected to point 1
      or if no more lines exist to connect new
```

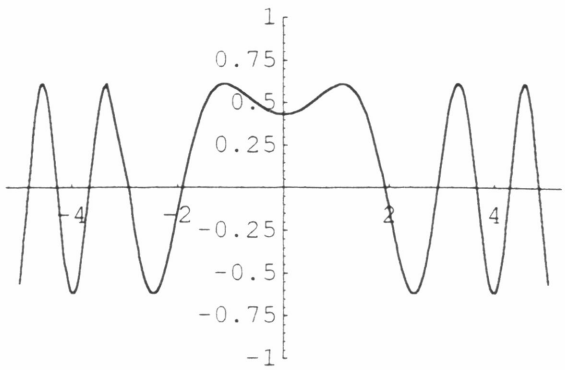
```
   points to those already connected to point 1,
   quit now.                                              *)
 If[ ((Length[value[[2]]] != p) || (value[[1]] != 0)),
   Block[ {i,j},
   (* Take a step. *)
   i = initi + inci;
   j = initj + incj;
   If[ i==initi, k=j, k=i ];
   (* Did we step off the graph?  If so, quit. *)
   If[ ((i > j) && (i <= p) && (j <= p)),
      (* If we've already visited this point or
         if this point is not directly connected
         to the point we just came from, keep
         walking in the same direction.          *)
      If[ ((G[[i,j]] != 0) &&
          (Length[Intersection[{k},newvalue[[2]]]]==0)),
         (* Keep track of how many lines are left
            to connect to new points.             *)
         Block[ {}, newvalue[[1]] -= G[[i,j]];
         (* We have found a new point not
            previously known to be connected to
            point 1!  First add this point to our
            list.  Next, if we were walking
            horizontally across the graph, now walk
            vertically down.  If we were walking
            down the graph, start walking left --
            when that's done, walk to the right
            from this same spot.                  *)
         If[ incj != 0,
            Block[ {},
            AppendTo[ newvalue[[2]], j ];
            newvalue=Gwalk[G,p,q,i,j,-1,0,newvalue];
            newvalue=Gwalk[G,p,q,i,j,1,0,newvalue];
            newvalue=Gwalk[G,p,q,i,j,inci,incj,
                           newvalue]],
            Block[ {},
            AppendTo[ newvalue[[2]], i ];
            newvalue=Gwalk[G,p,q,i,j,0,1,newvalue];
            newvalue=Gwalk[G,p,q,i,j,inci,incj,
                           newvalue]]
         ]],
      newvalue=Gwalk[G,p,q,i,j,inci,incj,newvalue]
      ]
   ]]
];
(* We're done walking in this direction, at least. *)
newvalue ]
```

■ **Supporting code taken from S. A. Fulling, I. Borosh, Cataloguing General Graphs by Point and Line Spectra. (In Progress)**

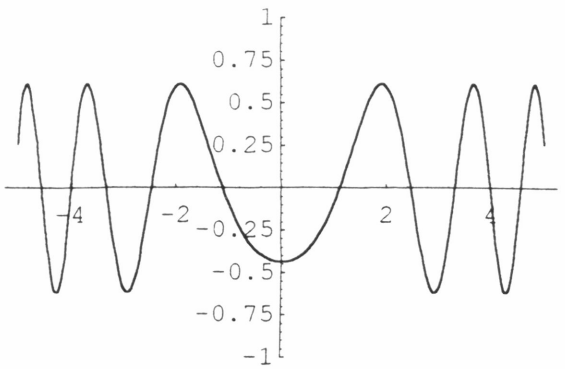
```
Needs["DiscreteMath`Combinatorica`"]

transposition[a_,b_,a_] := b
transposition[a_,b_,b_] := a
transposition[a_,b_,c_] := c
transposition[a_,b_] := transposition[a,b,#]&
listPerm[f_,l_List] := Array[l[[f[#]]]&, Length[l]]
rowPerm[f_,m_] := listPerm[f,m] /; MatrixQ[m]
columnPerm[f_,m_] := Map[listPerm[f,#]&, m] /; MatrixQ[m]
basisPerm[f_,m_] := MatrixForm[rowPerm[f,columnPerm[f,m]]]
tr[a_,b_,m_] := basisPerm[transposition[a,b], m]

adjMat[l_List, j_] := Block[{m, jj, i, r, c, hold},
   jj=j(j+1)/2;
   r=1;
   c=j;
   Do[
      m[r,c] = l[[i]];
      m[c,r] = m[r,c];
      If[r==1,
         hold=c; c=j; r=j-hold+2,
         r=r-1; c=c-1],
    {i, jj}];
   Array[m, {j,j}]]

labGraphs[j_, k_] := Map[MatrixForm[adjMat[#,j]]&,
                         Compositions[k,j(j+1)/2]]
```

■ An Example of How To Use The Code

```
v[x_] := x^2

Kcalc[hbar_,m_,d_,y0_,x0_,t_,fn_]:=
      Sqrt[m / (2 Pi I hbar t)] *
      Exp[I m (x0-y0)^2 / (2 hbar t)] *
      Exp[fn[y0,x0,t,m,hbar,1]]
Kothercalc[hbar_,m_,d_,y0_,x0_,t_,fn_]:=
      Sqrt[m / (2 Pi I hbar t)] *
      Exp[I m (x0-y0)^2 / (2 hbar t)] *
      (1+fn[y0,x0,t,m,hbar,1])
Ktrue[h_,m_,d_,y_,x_,t_,om_] :=
      Exp[I m om /(2 h Sin[om t]) *
      ((x^2 + y^2)Cos[ om t] - 2x y)] *
      Sqrt[m om/(2 Pi I h Sin[om t])] *
      (m/(2Pi I h t))^((d-1)/2)

RePKtrue = Plot[Re[Ktrue[1,2,1,0,x,1,1]],{x,-5,5},
                                  PlotRange->{-1,1}];
```




```
ImPKtrue = Plot[Im[Ktrue[1,2,1,0,x,1,1]],{x,-5,5},
                                  PlotRange->{-1,1}];
```

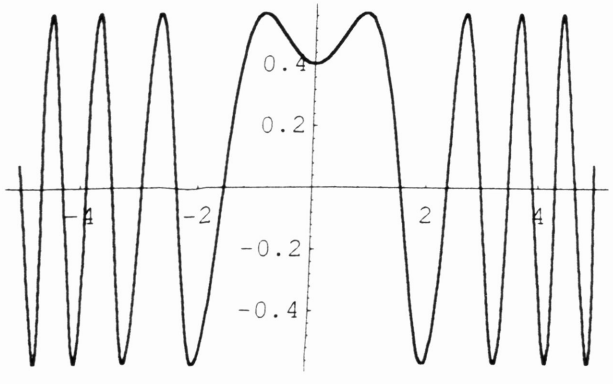


```
condfn[p_,q_] := If[ q <= 2, If[ p <= q+1, 1, 0 ], 0 ]
LExpSeries[ condfn ]
```

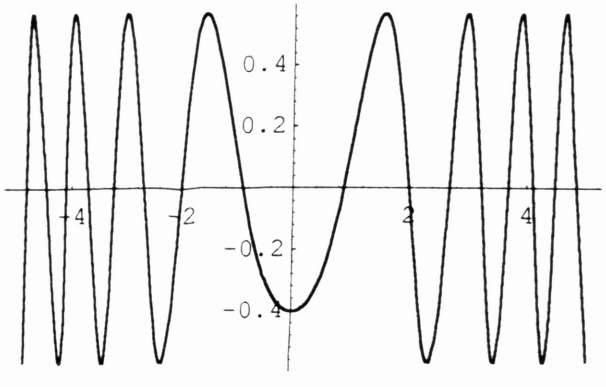
$$\frac{\lambda t^2}{3m}$$

$$-\frac{i}{16}\left(\frac{\lambda t}{\hbar}\left(\frac{(x_0 - y_0)^2}{3} + (x_0 - y_0)\,y_0 + y_0^2\right)\right)$$

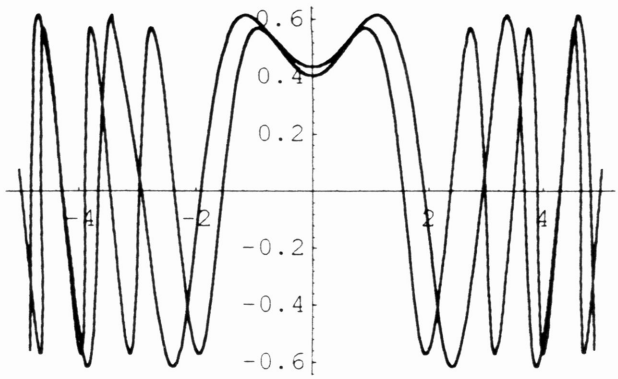
```
Simplify[%82] //InputForm
((lambda*t^2)/(3*m) -
    (I/3*lambda*t*(x0^2 + x0*y0 + y0^2))/hbar)/16
ftryme[y0_,x0_,t_,m_,hbar_,lambda_] :=
((lambda*t^2)/(3*m) -
    (I/3*lambda*t*(x0^2 + x0*y0 + y0^2))/hbar)/16
CalcRe = Plot[Re[Kcalc[1,2,1,0,x,1,ftryme]],{x,-5,5}];
```



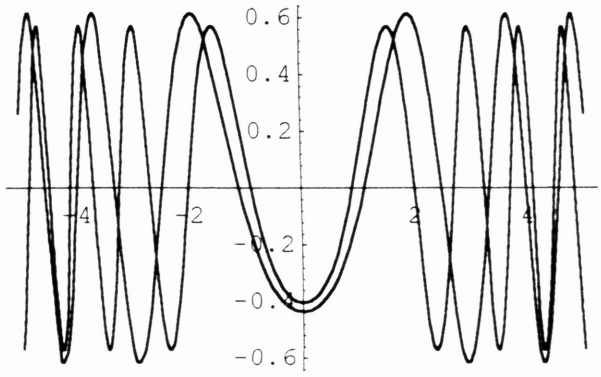

```
CalcIm = Plot[Im[Kcalc[1,2,1,0,x,1,ftryme]],{x,-5,5}];
```



```
Show[ CalcRe, RePKtrue ];
```




```
Show[ CalcIm, ImPKtrue ];
```



Appendix D

Plots of Numerical Results

Neat Graphs with h = 1

Quadratic, Sinusoidal, and Constant Magnetic Field Cases

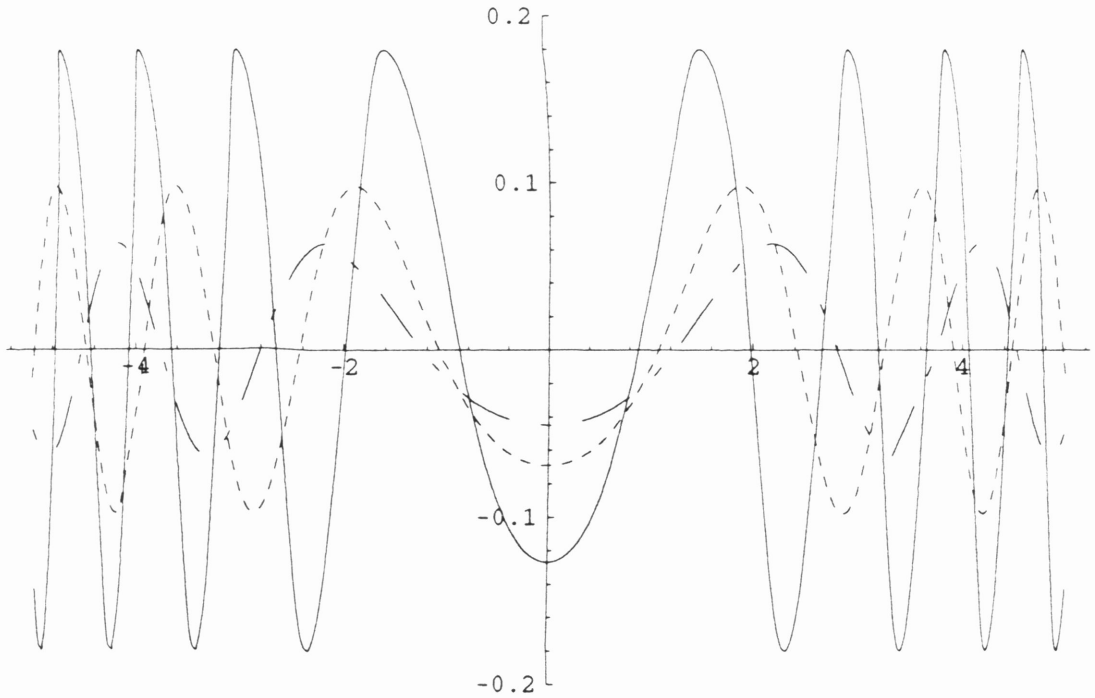
h = 1, with y_1 = 0, 1, 2 cases

We begin with the definition of the free propagator (QSE 1.5). This time, we set h to 1. We also set a temporary value for mass m of 2. Default values used in these calculations are y = {0 or 1 or 2,0,0}, x = {(variable),0,0}.
For y_1 = 0:

```
h = 1
m = 2
d = 3
K0[x_,t_]:=Exp[I m x^2 / (2 h t)] *
            (m / (2 Pi I h t))^(d/2)
```

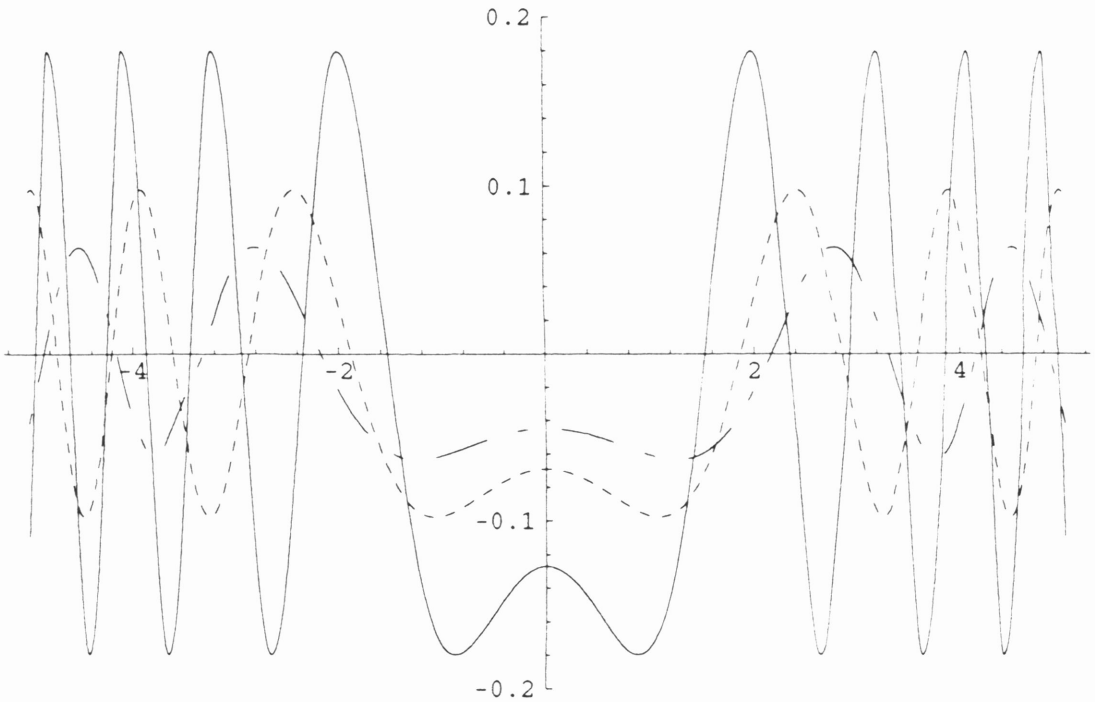
First the Real part:

```
Show[ Plot[ Re[K0[x,1]], {x,-5,5} ],
      Plot[ Re[K0[x,1.5]], {x,-5,5},
         PlotStyle -> Dashing[{.01,.01}] ],
      Plot[ Re[K0[x,2]], {x,-5,5},
         PlotStyle -> Dashing[{.01,.03,.05,.03}] ],
      PlotRange -> {-.2,.2} ]
```



Then the Imaginary part:

```
Show[ Plot[ Im[K0[x,1]], {x,-5,5} ],
      Plot[ Im[K0[x,1.5]], {x,-5,5},
         PlotStyle -> Dashing[{.01,.01}] ],
      Plot[ Im[K0[x,2]], {x,-5,5},
         PlotStyle -> Dashing[{.01,.03,.05,.03}] ],
      PlotRange -> {-.2,.2} ]
```



And we would see that for m = 20, plotting K_0 creates a huge blob of carbon, thus it is not displayed here.

For $y_1 = 1$, these plots of K0 would be shifted along the horizontal axis 1 unit to the right. And for $y_1 = 2$, these plots would be shifted 2 units.

▩ Constant Magnetic Field Case (with $y_1 = 0$ and h = 1)

v = 0 a = { -1/2 y, 1/2 x, 0 }

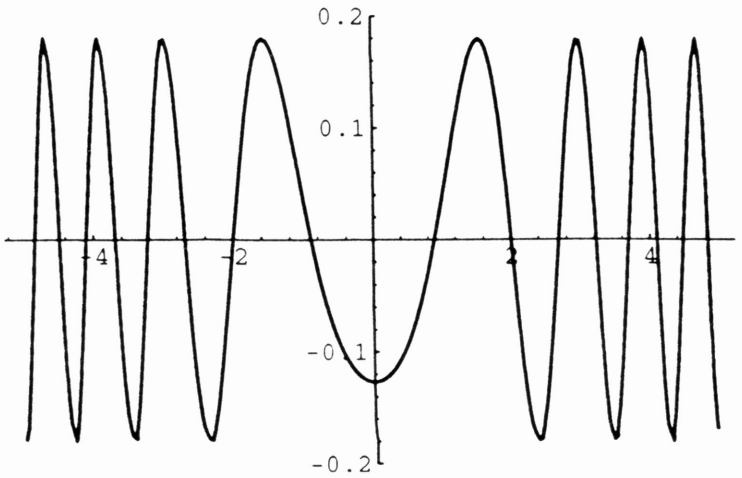
```
h = 1
m = 2
d = 3
K0[x_,t_]:= Exp[I m x^2 / (2 h t)] *
            (m / (2 Pi I h t))^(d/2)

T2t100 = ReadList["./Csource/Flat/ConstMag/t2t100",
                  {Number,Number,Number,Number} ]
T2t150 = ReadList["./Csource/Flat/ConstMag/t2t150",
                  {Number,Number,Number,Number} ]
T2t200 = ReadList["./Csource/Flat/ConstMag/t2t200",
                  {Number,Number,Number,Number} ]

T1t100 = ReadList["./Csource/Flat/ConstMag/t1t100",
                  {Number,Number} ]
T1t150 = ReadList["./Csource/Flat/ConstMag/t1t150",
                  {Number,Number} ]
T1t200 = ReadList["./Csource/Flat/ConstMag/t1t200",
                  {Number,Number} ]

Jt100 = ReadList["./Csource/Flat/ConstMag/jt100", Number ]
Jt150 = ReadList["./Csource/Flat/ConstMag/jt150", Number ]
Jt200 = ReadList["./Csource/Flat/ConstMag/jt200", Number ]
```

■ Case 1: t = 1.00

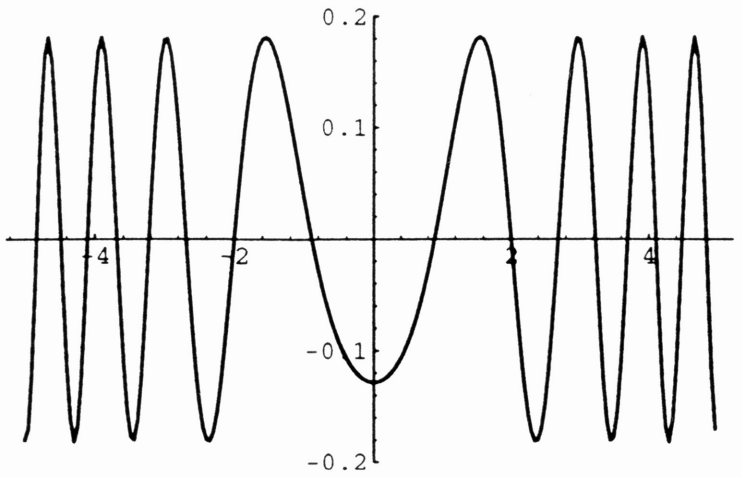
```
T2 = Table[ (I h)^-2 * T2t100[[i]][[1]]
          + (I h)^-1 * T2t100[[i]][[2]]
          + T2t100[[i]][[3]]
          + (I h) * T2t100[[i]][[4]],
     {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t100[[i]][[1]]
          + T1t100[[i]][[2]],
     {i,1,200} ]

J = Jt100
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,1] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

There will be no plots of K0ExpJ for any of the constant magnetic field cases looked at since J = 0 for all x and t. The plot of K0ExpJ is the same as the plot of K0.

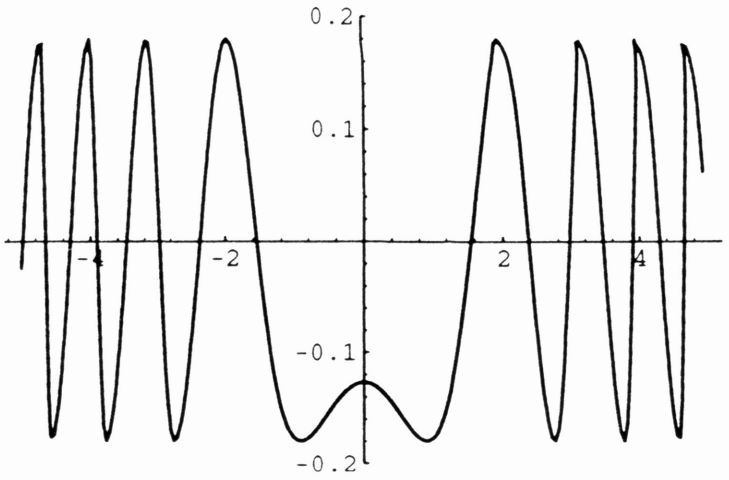
```
ListPlot[
   Table[ { (i-101)/20,
            Re[( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.2,.2} ]
```



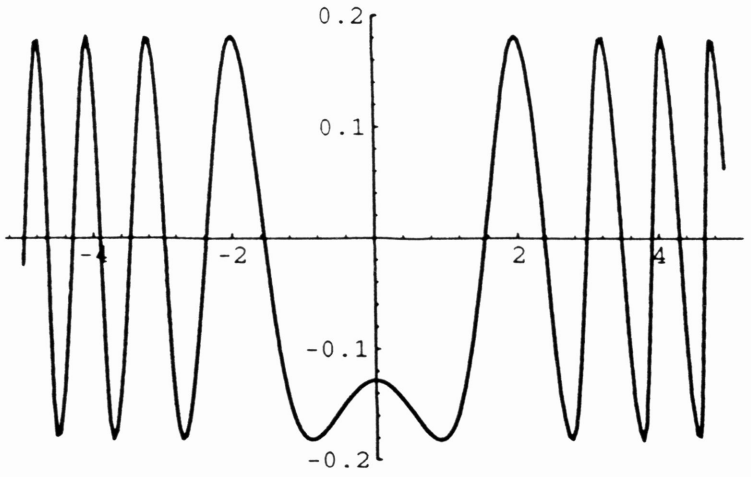

```
ListPlot[
   Table[ { (i-101)/20,
            Re[( K0ExpJ[[i]] *
               Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.2,.2} ]
```



```
ListPlot[
   Table[ { (i-101)/20,
            Im[( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.2,.2} ]
```

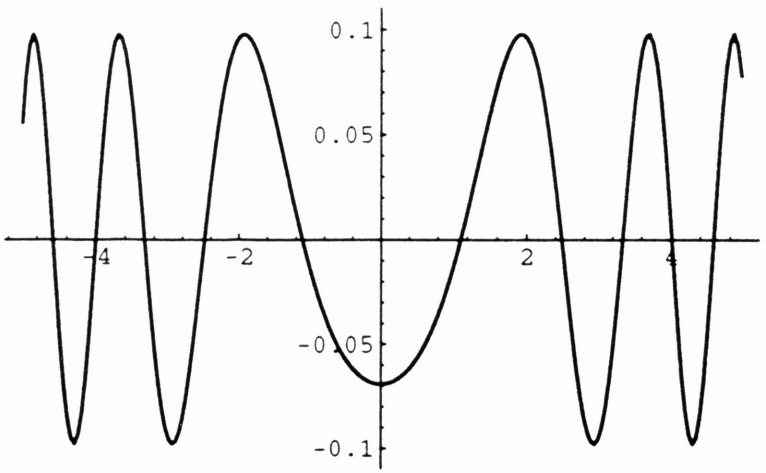



```
ListPlot[
   Table[ { (i-101)/20,
            Im[( K0ExpJ[[i]] *
               Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.2,.2} ]
```

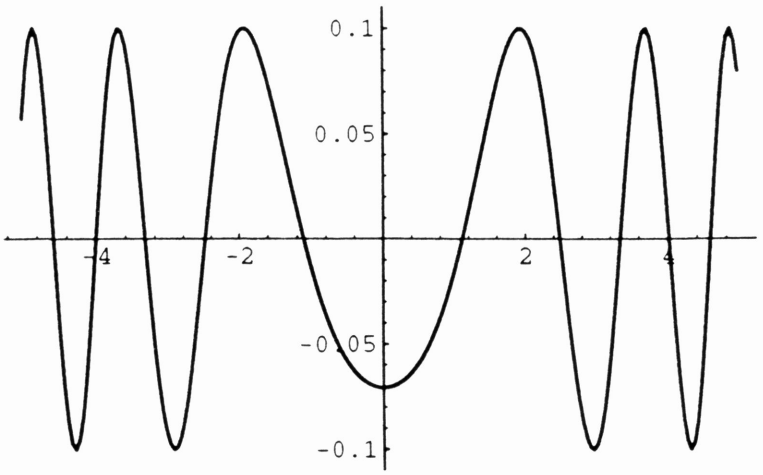


■ Case 2: t = 1.50

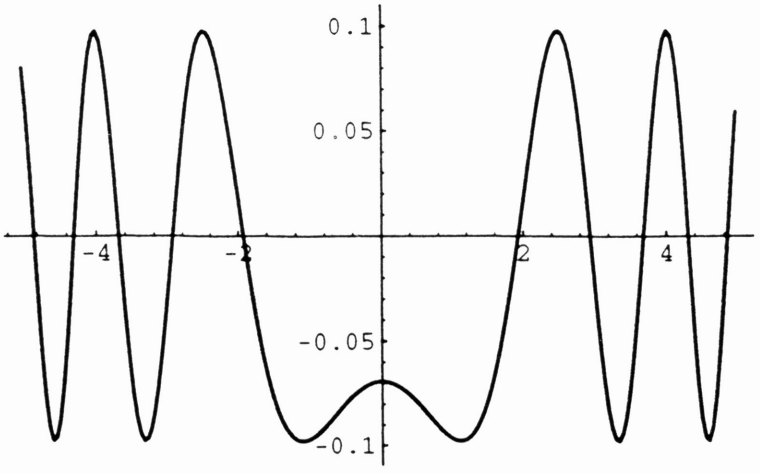
```
T2 = Table[ (I h)^-2 * T2t150[[i]][[1]]
            + (I h)^-1 * T2t150[[i]][[2]]
            + T2t150[[i]][[3]]
            + (I h) * T2t150[[i]][[4]],
      {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t150[[i]][[1]]
             + T1t150[[i]][[2]],
      {i,1,200} ]

J = Jt150
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,1.5] *
                  Exp[ J[[i]] / (I h) ] ),
          {i,1,200} ]

ListPlot[
   Table[ { (i-101)/20,
            Re[( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.11,.11} ]
```



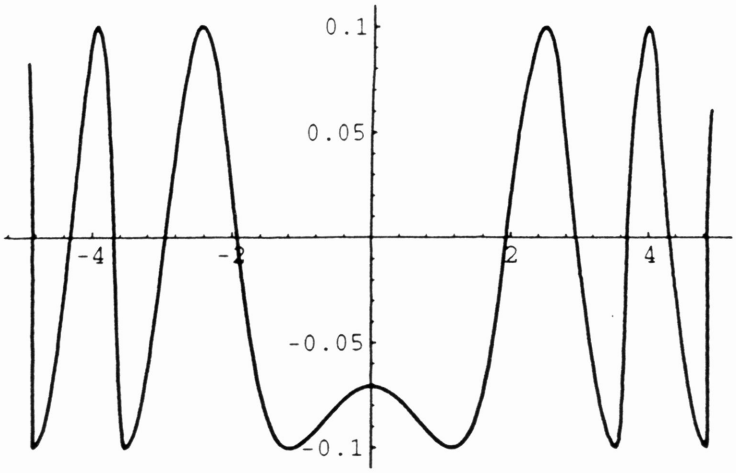
```
ListPlot[
   Table[ { (i-101)/20,
            Re[( K0ExpJ[[i]] *
               Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.11,.11} ]
```




```
ListPlot[
   Table[ { (i-101)/20,
            Im[( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.11,.11} ]
```



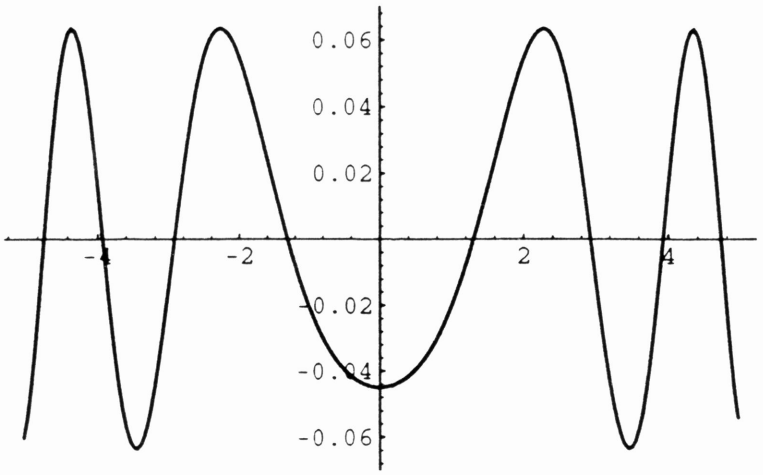
```
ListPlot[
   Table[ { (i-101)/20,
            Im[( K0ExpJ[[i]] *
               Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.11,.11} ]
```



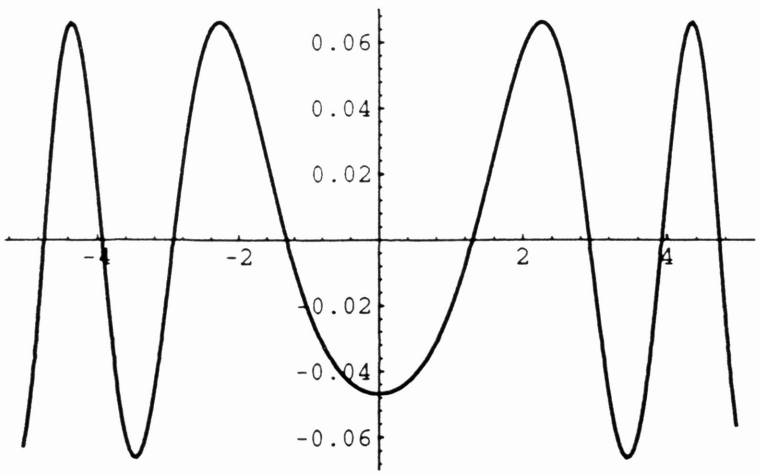
■ Case 3: t = 2.00

```
T2 = Table[ (I h)^-2 * T2t200[[i]][[1]]
          + (I h)^-1 * T2t200[[i]][[2]]
          + T2t200[[i]][[3]]
          + (I h) * T2t200[[i]][[4]],
     {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t200[[i]][[1]]
          + T1t200[[i]][[2]],
     {i,1,200} ]

J = Jt200
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,2] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

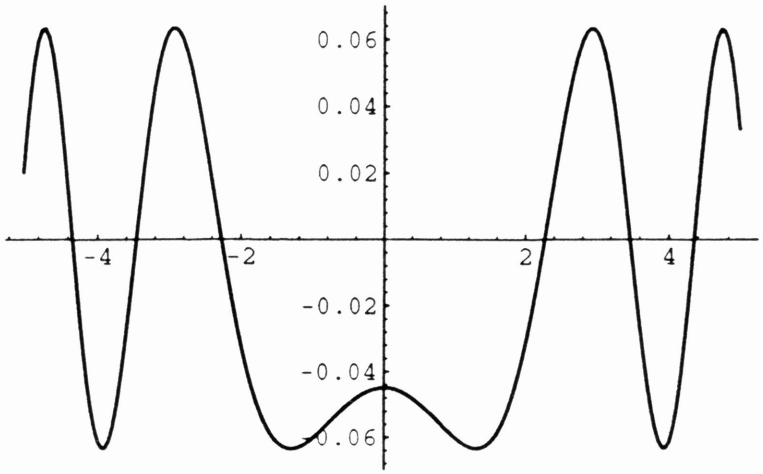
```
ListPlot[
   Table[ { (i-101)/20,
            Re[( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.07,.07} ]
```



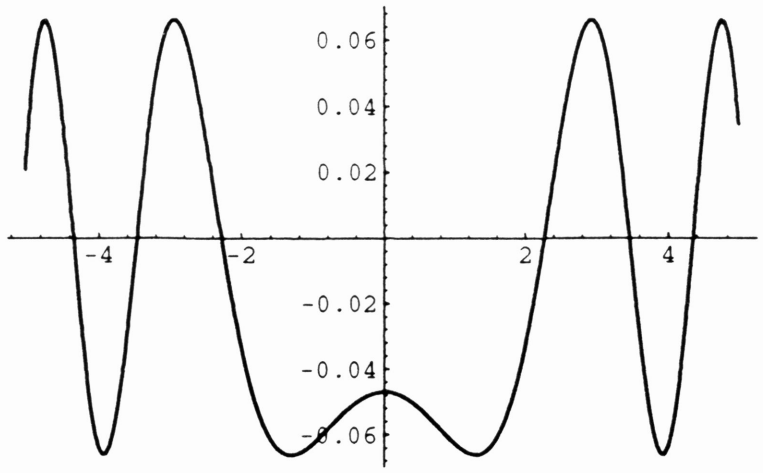

```
ListPlot[
   Table[ { (i-101)/20,
            Re[( K0ExpJ[[i]] *
               Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.07,.07} ]
```



```
ListPlot[
   Table[ { (i-101)/20,
            Im[( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.07,.07} ]
```




```
ListPlot[
   Table[ { (i-101)/20,
            Im[( K0ExpJ[[i]] *
               Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
   {i,1,200} ],
   PlotJoined->True, AxesOrigin->{0,0},
   PlotRange->{-.07,.07} ]
```



▩ Sinusoidal Case (with $y_1 = 0$ and h = 1)

$$v = 0 \qquad a = \{ 0,\ 2\,\mathrm{Sin}[x]^{t},\ 0 \}$$

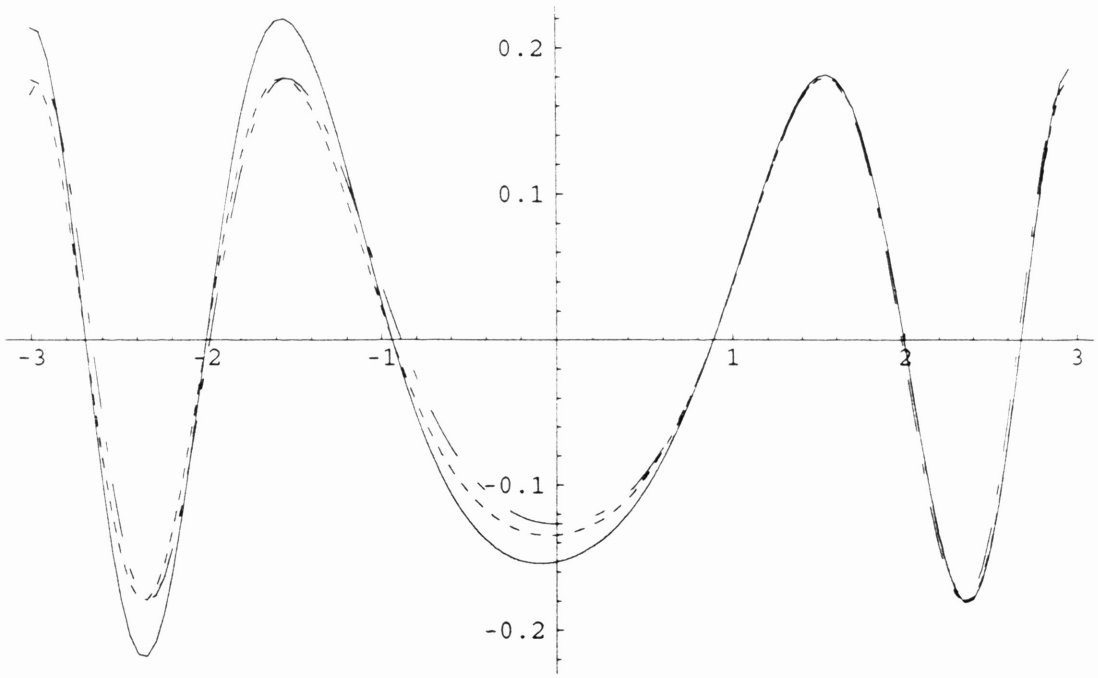
```
h = 1
m = 2
d = 3
K0[x_,t_]:=Exp[I m x^2 / (2 h t)] *
           (m / (2 Pi I h t))^(d/2)

T2t100 = ReadList["./Csource/Flat/Sinus1/t2t100",
                  {Number,Number,Number,Number} ]
T2t150 = ReadList["./Csource/Flat/Sinus1/t2t150",
                  {Number,Number,Number,Number} ]
T2t200 = ReadList["./Csource/Flat/Sinus1/t2t200",
                  {Number,Number,Number,Number} ]

T1t100 = ReadList["./Csource/Flat/Sinus1/t1t100",
                  {Number,Number} ]
T1t150 = ReadList["./Csource/Flat/Sinus1/t1t150",
                  {Number,Number} ]
T1t200 = ReadList["./Csource/Flat/Sinus1/t1t200",
                  {Number,Number} ]

Jt100 = ReadList["./Csource/Flat/Sinus1/jt100", Number ]
Jt150 = ReadList["./Csource/Flat/Sinus1/jt150", Number ]
Jt200 = ReadList["./Csource/Flat/Sinus1/jt200", Number ]
```

■ Case 1: t = 1.00

```
T2 = Table[ (I h)^-2 * T2t100[[i]][[1]]
          + (I h)^-1 * T2t100[[i]][[2]]
          + T2t100[[i]][[3]]
          + (I h) * T2t100[[i]][[4]],
     {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t100[[i]][[1]]
          + T1t100[[i]][[2]],
     {i,1,200} ]

J = Jt100
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,1] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

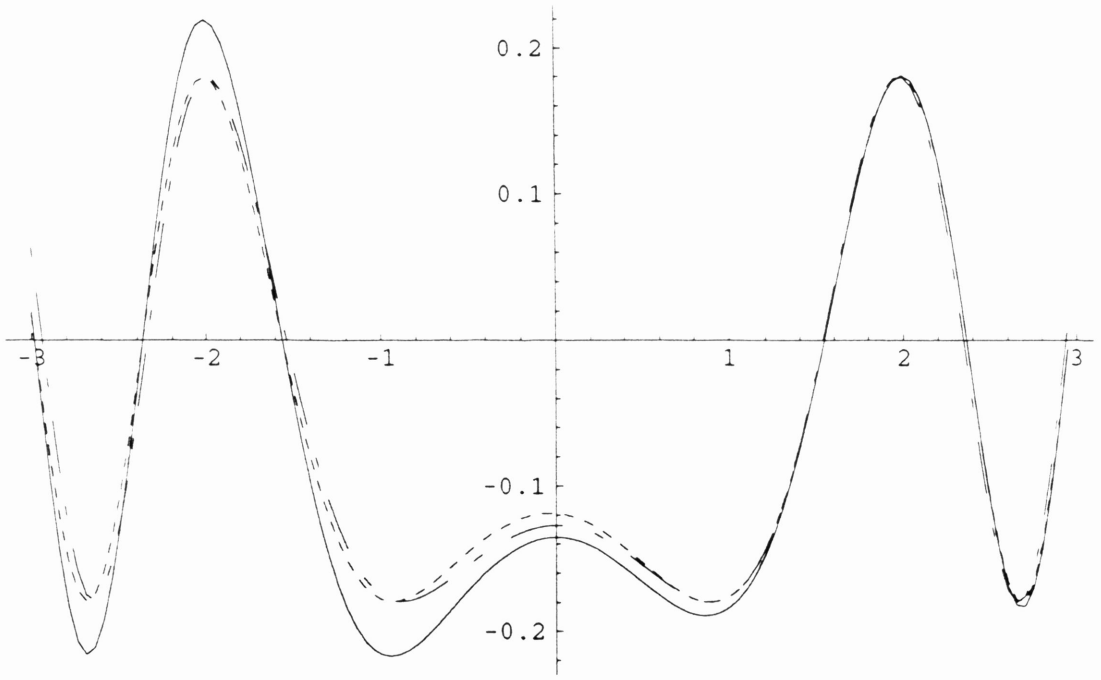
☐ Comparison of the Real parts of $K_0 Exp[J]$, $K_0 Exp[J] Exp[1/m\ J1]$, and $K_0 Exp[J] Exp[1/m\ J1 + 1/m^2\ J2]$

```
Show[ ListPlot[ Table[ {(i-101)/20, Re[K0ExpJ[[i]]]},
                {i,41,160} ], PlotJoined->True,
                AxesOrigin->{0,0},
                PlotStyle->Dashing[{.01,.03,.05,.03}] ],
      ListPlot[ Table[ { (i-101)/20, Re[
                ( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
                {i,41,160} ],
                PlotJoined->True, AxesOrigin->{0,0},
                PlotStyle->Dashing[{.01,.01}] ],
      ListPlot[ Table[ { (i-101)/20,
                Re[( K0ExpJ[[i]] *
                Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
                {i,41,160} ],
                PlotJoined->True, AxesOrigin->{0,0} ],
      PlotRange->{-.23,.23} ]
```



☐ Comparison of the Imaginary parts of $K_0 Exp[\ J\ ]$, $K_0 Exp[\ J\ ] Exp[\ 1/m\ J1\ ]$, and $K_0 Exp[\ J\ ] Exp[\ 1/m\ J1 + 1/m^2\ J2\ ]$

```
Show[ ListPlot[ Table[ {(i-101)/20, Im[K0ExpJ[[i]]]},
                {i,41,160} ], PlotJoined->True,
                AxesOrigin->{0,0},
                PlotStyle->Dashing[{.01,.03,.05,.03}] ],
      ListPlot[ Table[ { (i-101)/20, Im[
                ( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
                {i,41,160} ],
                PlotJoined->True, AxesOrigin->{0,0},
                PlotStyle->Dashing[{.01,.01}] ],
      ListPlot[ Table[ { (i-101)/20,
                Im[( K0ExpJ[[i]] *
                Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
                {i,41,160} ],
                PlotJoined->True, AxesOrigin->{0,0} ],
      PlotRange->{-.23,.23} ]
```

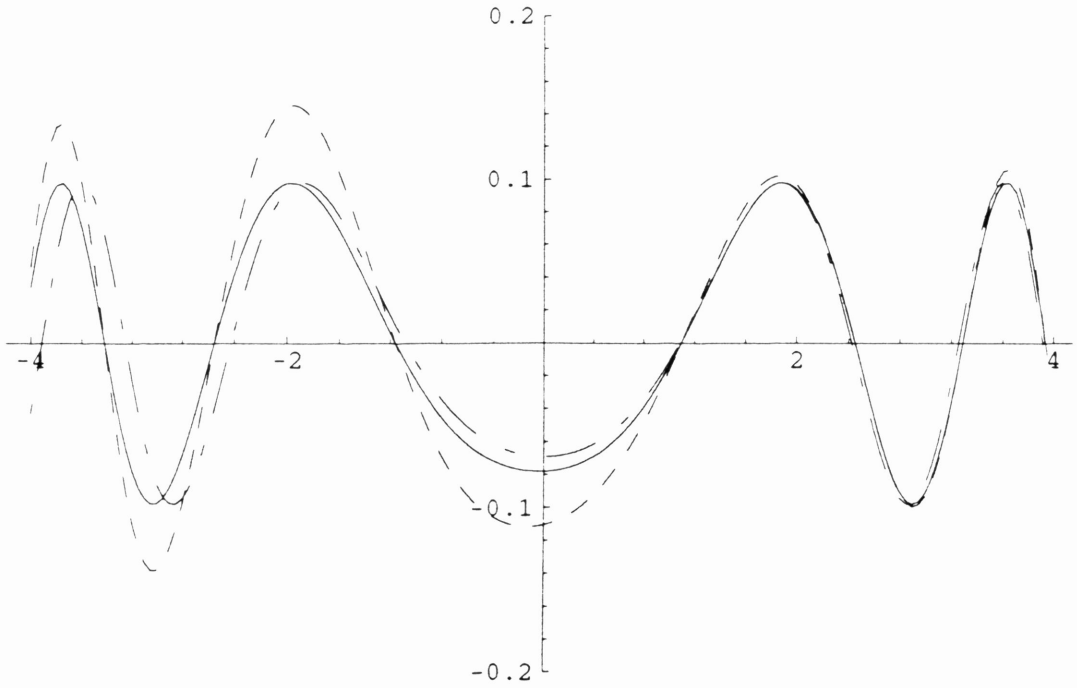


■ Case 2: t = 1.50

```
T2 = Table[ (I h)^-2 * T2t150[[i]][[1]]
            + (I h)^-1 * T2t150[[i]][[2]]
            + T2t150[[i]][[3]]
            + (I h) * T2t150[[i]][[4]],
     {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t150[[i]][[1]]
             + T1t150[[i]][[2]],
     {i,1,200} ]

J = Jt150
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,1.5] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

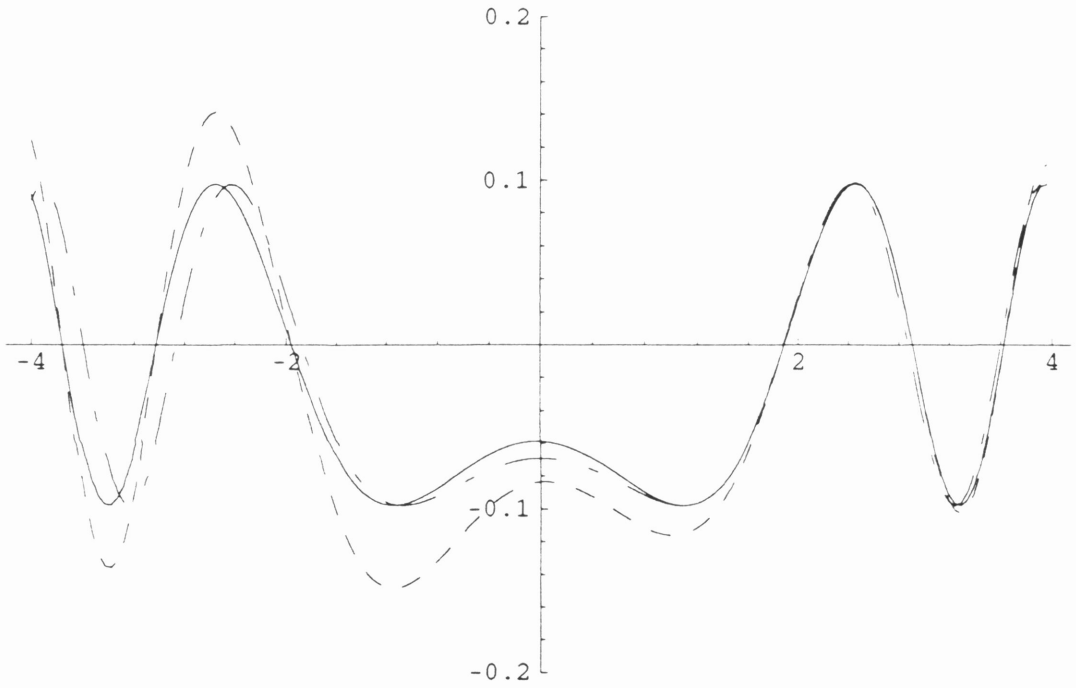
☐ Comparison of the Real parts of $K_0 Exp[J]$, $K_0 Exp[J] Exp[1/m\ J1]$, and $K_0 Exp[J] Exp[1/m\ J1 + 1/m^2\ J2]$

```
Show[ListPlot[Table[{(i-101)/20, Re[K0ExpJ[[i]] ]},
        {i,21,180}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
    ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,21,180}], PlotJoined->True, AxesOrigin->{0,0} ]
    ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,21,180}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.02,.02}]],
    PlotRange->{-.2,.2} ]
```



☐ Comparison of the Imaginary parts of $K_0Exp[\ J\ ]$, $K_0Exp[\ J\ ]Exp[\ 1/m\ J1\ ]$, and $K_0Exp[\ J\ ]Exp[\ 1/m\ J1 + 1/m^2\ J2\ ]$

```
Show[ListPlot[Table[{(i-101)/20, Im[K0ExpJ[[i]] ]},
        {i,21,180}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
     ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,21,180}], PlotJoined->True, AxesOrigin->{0,0} ]
     ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,21,180}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.02,.01}]],
     PlotRange->{-.2,.2} ]
```

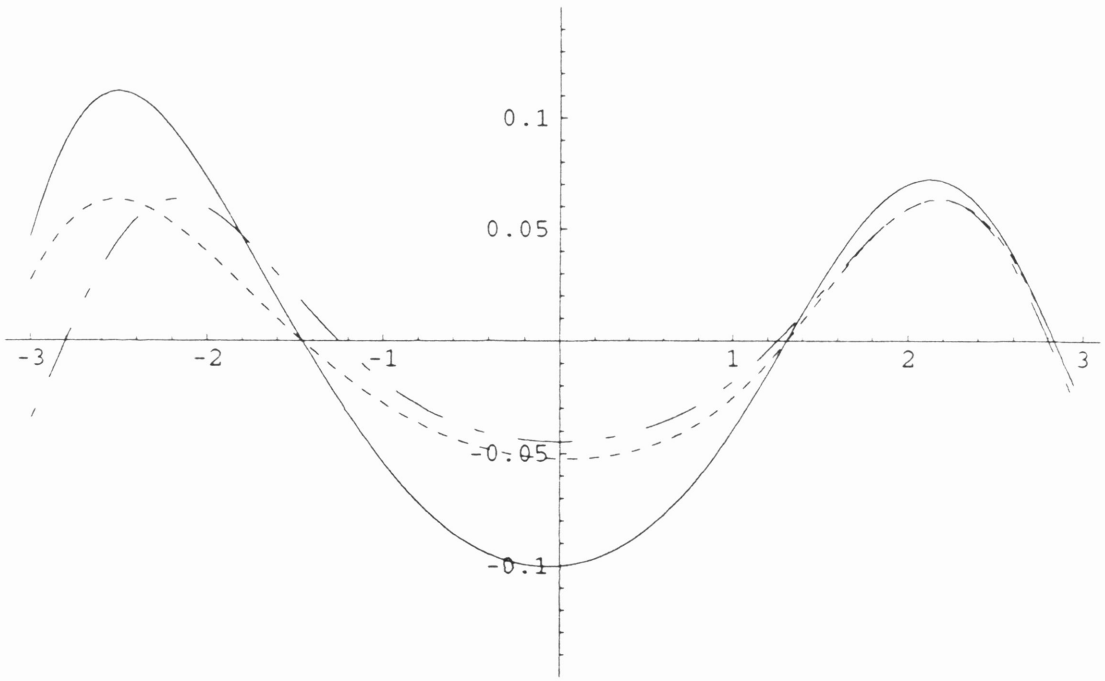


■ Case 3: t = 2.00

```
T2 = Table[ (I h)^-2 * T2t200[[i]][[1]]
            + (I h)^-1 * T2t200[[i]][[2]]
            + T2t200[[i]][[3]]
            + (I h) * T2t200[[i]][[4]],
     {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t200[[i]][[1]]
             + T1t200[[i]][[2]],
     {i,1,200} ]

J = Jt200
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,2] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

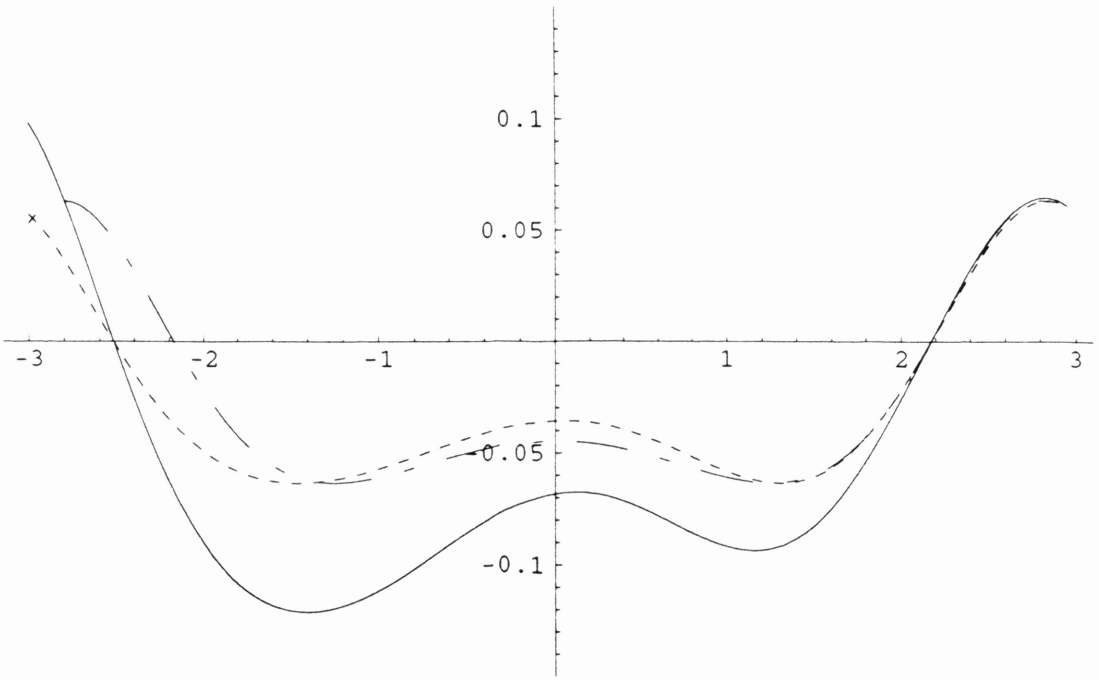
☐ Comparison of the Real parts of $K_0 Exp[\ J\ ]$, $K_0 Exp[\ J\ ] Exp[\ 1/m\ J1\ ]$, and $K_0 Exp[\ J\ ] Exp[\ 1/m\ J1 + 1/m^2\ J2\ ]$

```
Show[ ListPlot[ Table[ {(i-101)/20, Re[K0ExpJ[[i]]]},
                {i,41,160} ], PlotJoined->True,
                AxesOrigin->{0,0},
                PlotStyle->Dashing[{.01,.03,.05,.03}] ],
      ListPlot[ Table[ { (i-101)/20, Re[
                ( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
                {i,41,160} ],
                PlotJoined->True, AxesOrigin->{0,0},
                PlotStyle->Dashing[{.01,.01}] ],
      ListPlot[ Table[ { (i-101)/20,
                Re[( K0ExpJ[[i]] *
                Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
                {i,41,160} ],
                PlotJoined->True, AxesOrigin->{0,0} ],
      PlotRange->{-.15,.15} ]
```



☐ Comparison of the Imaginary parts of $K_0Exp[\ J\ ]$, $K_0Exp[\ J\ ]Exp[\ 1/m\ J1\ ]$, and $K_0Exp[\ J\ ]Exp[\ 1/m\ J1 + 1/m^2\ J2\ ]$

```
Show[ ListPlot[ Table[ {(i-101)/20, Im[K0ExpJ[[i]]]}, ——— - ——— -
                {i,41,160} ], PlotJoined->True,
                AxesOrigin->{0,0},
                PlotStyle->Dashing[{.01,.03,.05,.03}] ],
      ListPlot[ Table[ { (i-101)/20, Im[
                ( K0ExpJ[[i]] * Exp[ 1/m J1[[i]] ] )] },
                {i,41,160} ],
                PlotJoined->True, AxesOrigin->{0,0},
                PlotStyle->Dashing[{.01,.01}] ],
      ListPlot[ Table[ { (i-101)/20,
                Im[( K0ExpJ[[i]] *
                Exp[ 1/m J1[[i]] + 1/m^2 J2[[i]] ] )] },
                {i,41,160} ],
                PlotJoined->True, AxesOrigin->{0,0} ],
      PlotRange->{-.15,.15} ]
```



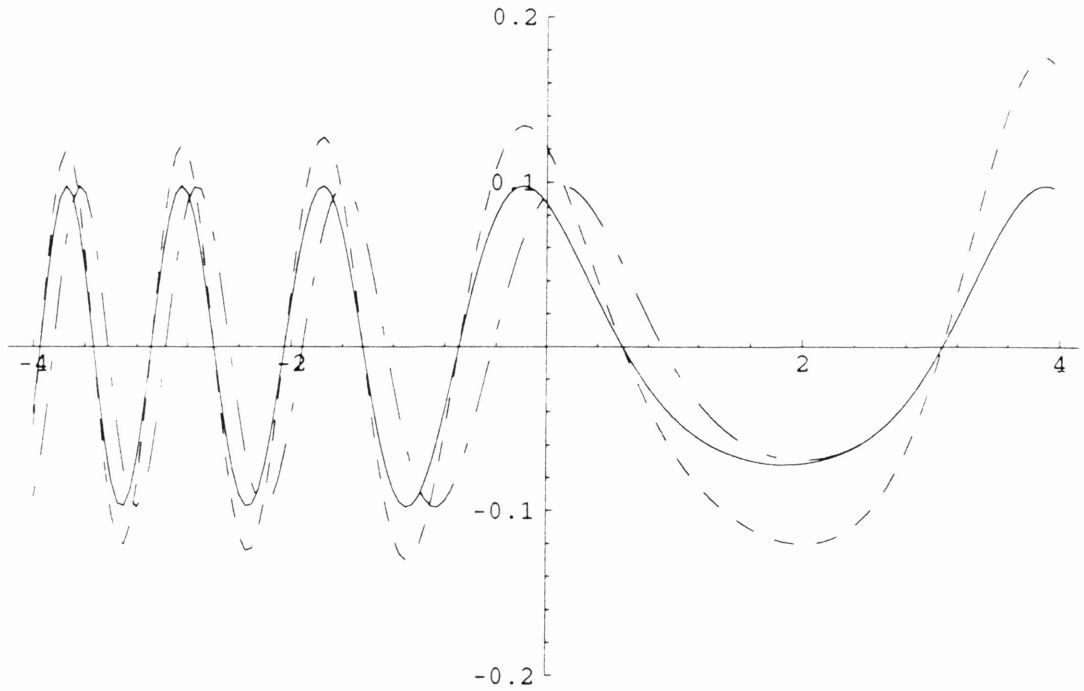
Sinusoidal Case (with $y_1 = 2$, $h = 1$)

Case 1: t = 1.50 and h = 1

```
h = 1
m = 2
d = 3
K0[x_,t_]:=Exp[I m (x-2)^2 / (2 h t)] *
           (m / (2 Pi I h t))^(d/2)

T2 = Table[ (I h)^-2 * T2t150[[i]][[1]]
          + (I h)^-1 * T2t150[[i]][[2]]
          + T2t150[[i]][[3]]
          + (I h) * T2t150[[i]][[4]],
     {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t150[[i]][[1]]
          + T1t150[[i]][[2]],
     {i,1,200} ]

J = Jt150
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,1.5] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

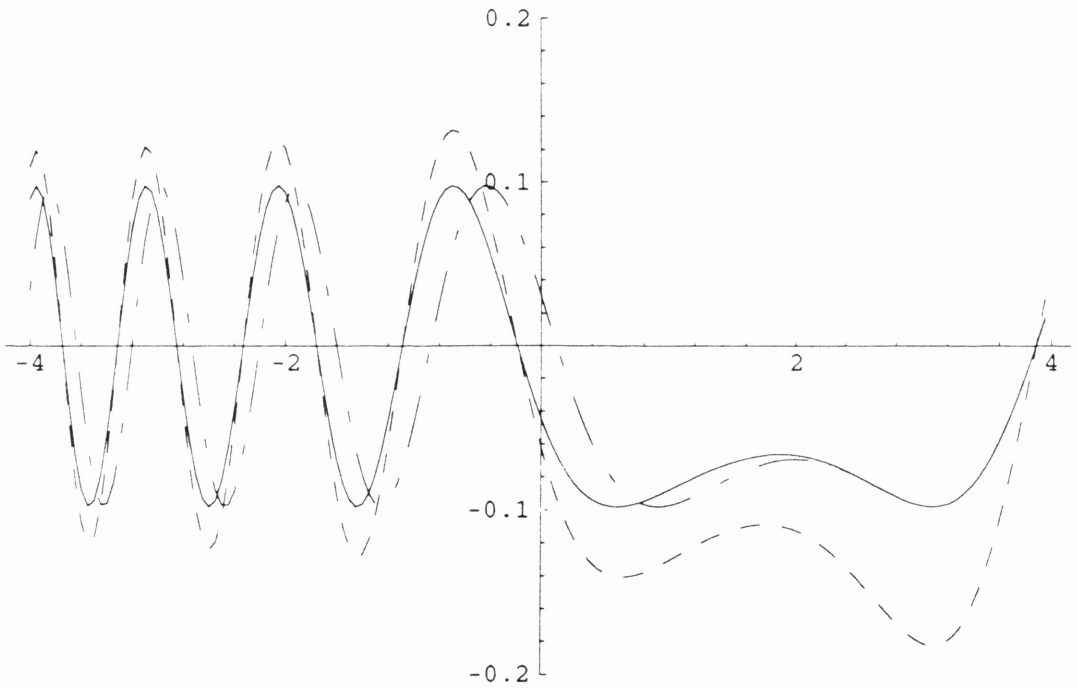
☐ Comparison of the Real parts of $K_0Exp[\ J\ ]$, $K_0Exp[\ J\ ]Exp[\ 1/m\ J1\ ]$, and $K_0Exp[\ J\ ]Exp[\ 1/m\ J1 + 1/m^2\ J2\ ]$

```
Show[ListPlot[Table[{(i-101)/20, Re[K0ExpJ[[i]] ]},
        {i,21,180}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
    ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,21,180}], PlotJoined->True, AxesOrigin->{0,0} ]
    ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,21,180}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.002,.01}]],
    PlotRange->{-.2,.2} ]
```



☐ Comparison of the Imaginary parts of K_0Exp[J], K_0Exp[J]Exp[1/m J1], and K_0Exp[J]Exp[1/m J1 + 1/m^2 J2]

```
Show[ListPlot[Table[{(i-101)/20, Im[K0ExpJ[[i]] ]},
        {i,21,180}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
    ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,21,180}], PlotJoined->True, AxesOrigin->{0,0} ]
    ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,21,180}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.002,.01}]],
    PlotRange->{-.2,.2} ]
```



▩ Quadratic Case (with $y_1 = 0$ and h = 1)

v = x^2 a = { 0, 0, 0 }

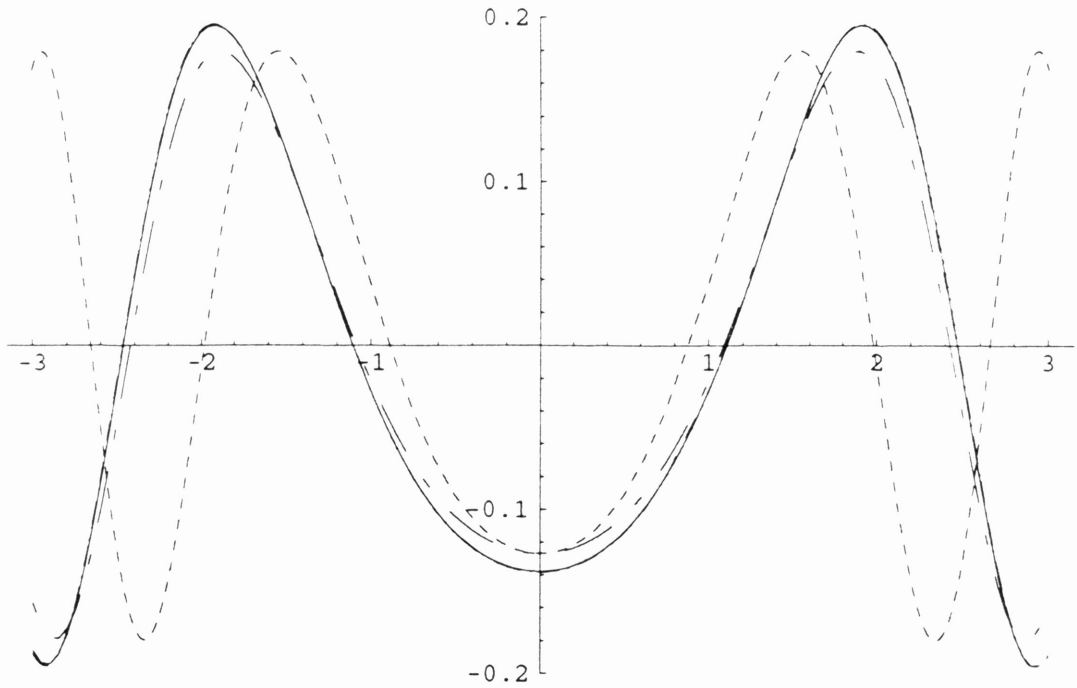
```
h = 1
m = 2
d = 3
K0[x_,t_]:=Exp[I m x^2 / (2 h t)] *
           (m / (2 Pi I h t))^(d/2)

T2t100 = ReadList["./Csource/Flat/Quad1/t2t100",
                  {Number,Number,Number,Number} ]
T2t150 = ReadList["./Csource/Flat/Quad1/t2t150",
                  {Number,Number,Number,Number} ]
T2t200 = ReadList["./Csource/Flat/Quad1/t2t200",
                  {Number,Number,Number,Number} ]

T1t100 = ReadList["./Csource/Flat/Quad1/t1t100",
                  {Number,Number} ]
T1t150 = ReadList["./Csource/Flat/Quad1/t1t150",
                  {Number,Number} ]
T1t200 = ReadList["./Csource/Flat/Quad1/t1t200",
                  {Number,Number} ]

Jt100 = ReadList["./Csource/Flat/Quad1/jt100", Number ]
Jt150 = ReadList["./Csource/Flat/Quad1/jt150", Number ]
Jt200 = ReadList["./Csource/Flat/Quad1/jt200", Number ]
```

■ Case 1: t = 1.00

```
T2 = Table[ (I h)^-2 * T2t100[[i]][[1]]
          + (I h)^-1 * T2t100[[i]][[2]]
          + T2t100[[i]][[3]]
          + (I h) * T2t100[[i]][[4]],
     {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t100[[i]][[1]]
          + T1t100[[i]][[2]],
     {i,1,200} ]

J = Jt100
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,1] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

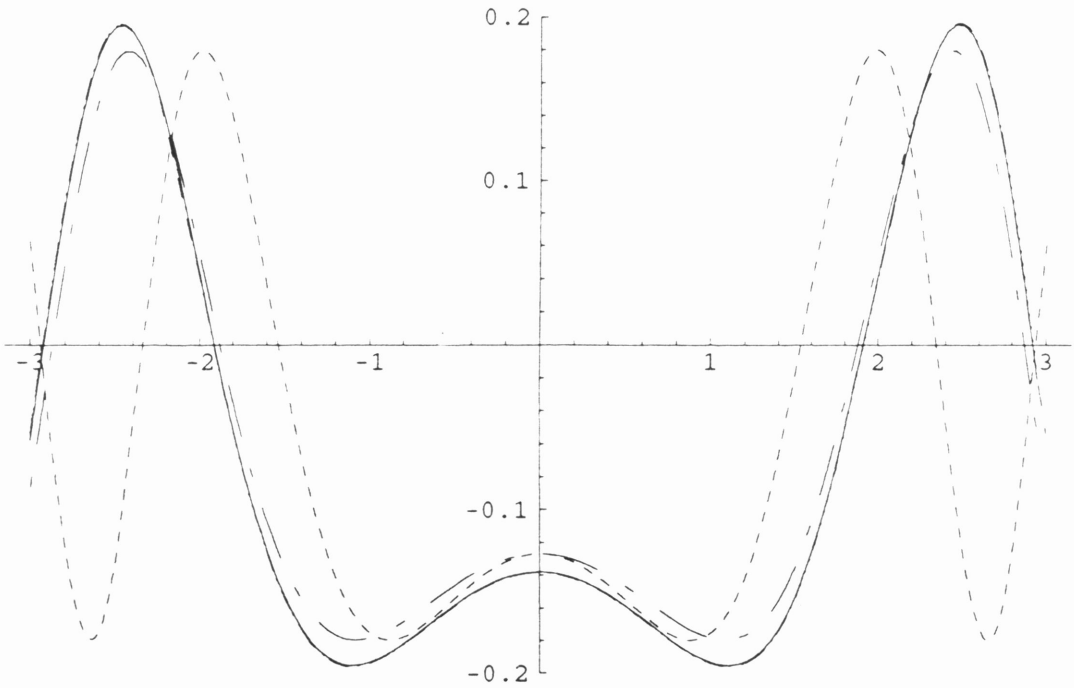
☐ Comparison of the Real parts of K_0, K_0Exp[J], K_0Exp[J]Exp[1/m J1], K_0Exp[J]Exp[1/m J1 + 1/m^2 J2] and the exact propagator

```
Show[Plot[Re[K0[x,t]], {x,-3,3},
        PlotStyle->Dashing[{.01,.01}]],
     ListPlot[Table[{(i-101)/20, Re[K0ExpJ[[i]] ]},
        {i,41,160}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
     ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,41,160}], PlotJoined->True, AxesOrigin->{0,0} ]
     ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,41,160}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.002,.01}]],
     Plot[Re[Ktrue[x,t]], {x,-3,3},
        PlotStyle->Dashing[{.02,.02}]],
     PlotRange->{-.2,.2} ]
```



☐ Comparison of the Imaginary parts of K_0, K_0Exp[J], K_0Exp[J]Exp[1/m J1], K_0Exp[J]Exp[1/m J1 + 1/m^2 J2] and the exact propagator

```
Show[Plot[Im[K0[x,t]], {x,-3,3},
        PlotStyle->Dashing[{.01,.01}]],
     ListPlot[Table[{(i-101)/20, Im[K0ExpJ[[i]] ]},
        {i,41,160}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
     ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,41,160}], PlotJoined->True, AxesOrigin->{0,0} ]
     ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,41,160}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.002,.01}]],
     Plot[Im[Ktrue[x,t]], {x,-3,3},
        PlotStyle->Dashing[{.02,.02}]],
     PlotRange->{-.2,.2} ]
```

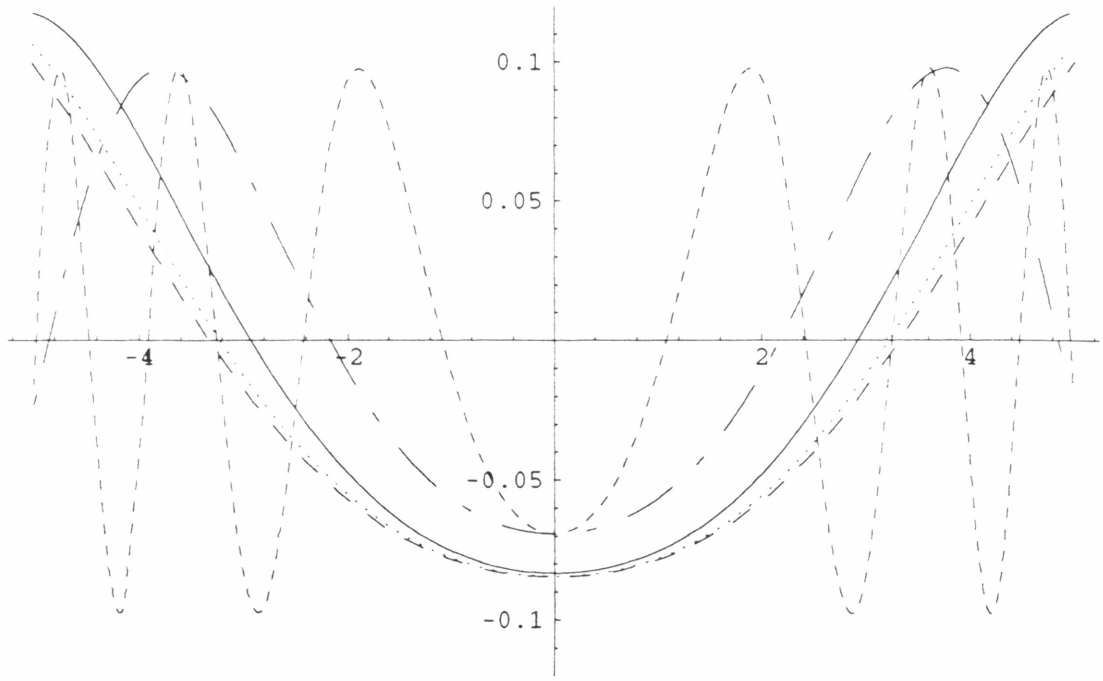


■ Case 2: t = 1.50

```
T2 = Table[ (I h)^-2 * T2t150[[i]][[1]]
            + (I h)^-1 * T2t150[[i]][[2]]
            + T2t150[[i]][[3]]
            + (I h) * T2t150[[i]][[4]],
     {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t150[[i]][[1]]
             + T1t150[[i]][[2]],
     {i,1,200} ]

J = Jt150
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,1.5] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

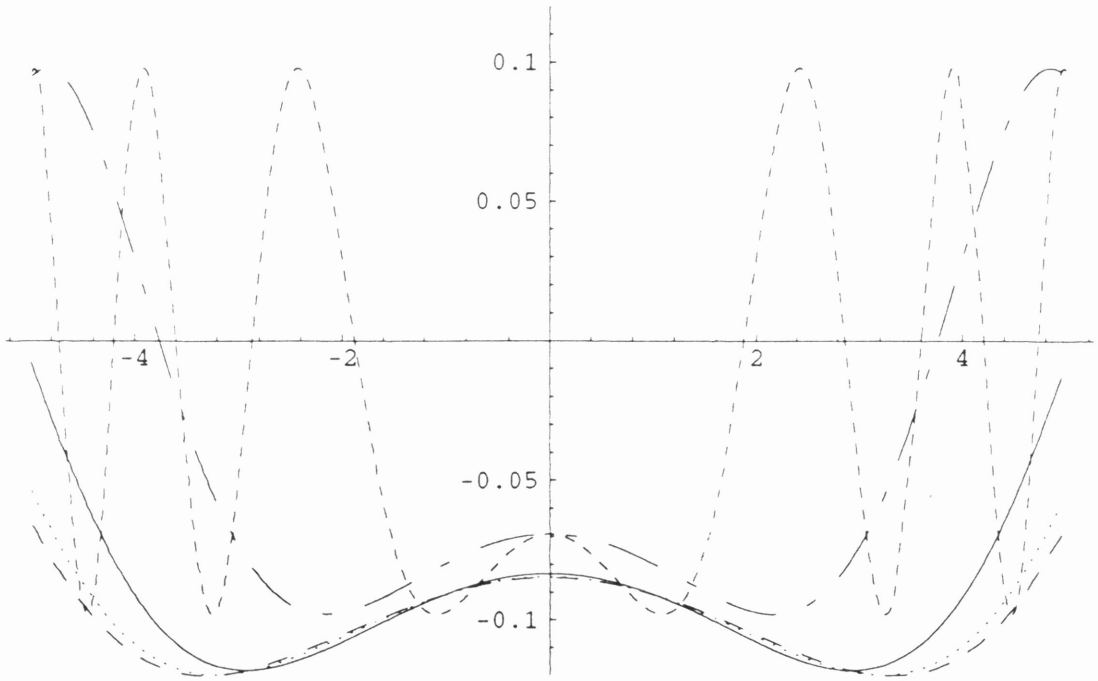
☐ Comparison of the Real parts of K_0, K_0Exp[J], K_0Exp[J]Exp[1/m J1], K_0Exp[J]Exp[1/m J1 + 1/m^2 J2] and the exact propagator

```
Show[Plot[Re[K0[x,t]], {x,-5,5},
        PlotStyle->Dashing[{.01,.01}]],
     ListPlot[Table[{(i-101)/20, Re[K0ExpJ[[i]] ]},
        {i,1,200}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
     ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,1,200}], PlotJoined->True, AxesOrigin->{0,0} ],
     ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,1,200}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.002,.01}]],
     Plot[Re[Ktrue[x,t]], {x,-5,5},
        PlotStyle->Dashing[{.02,.02}]],
     PlotRange->{-.12,.12} ]
```



☐ Comparison of the Imaginary parts of K_0, K_0Exp[J], K_0Exp[J]Exp[1/m J1], K_0Exp[J]Exp[1/m J1 + 1/m^2 J2] and the exact propagator

```
Show[Plot[Im[K0[x,t]], {x,-5,5},
        PlotStyle->Dashing[{.01,.01}]],
     ListPlot[Table[{(i-101)/20, Im[K0ExpJ[[i]] ]},
        {i,1,200}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
     ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,1,200}], PlotJoined->True, AxesOrigin->{0,0} ],
     ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,1,200}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.002,.01}]],
     Plot[Im[Ktrue[x,t]], {x,-5,5},
        PlotStyle->Dashing[{.02,.02}]],
     PlotRange->{-.12,.12} ]
```

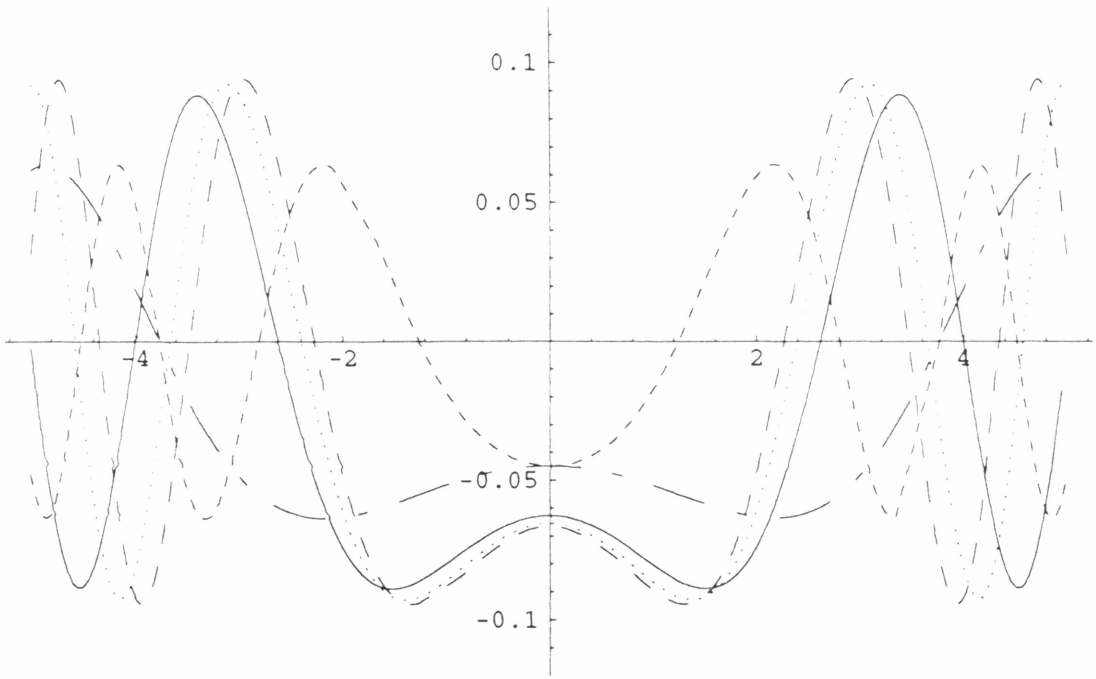


■ Case 3: t = 2.00

```
T2 = Table[ (I h)^-2 * T2t200[[i]][[1]]
            + (I h)^-1 * T2t200[[i]][[2]]
            + T2t200[[i]][[3]]
            + (I h) * T2t200[[i]][[4]],
      {i,1,200} ]

T1 = Table[ (I h)^-1 * T1t200[[i]][[1]]
            + T1t200[[i]][[2]],
      {i,1,200} ]

J = Jt200
J1 = T1
J2 = T2 - 1/2 T1^2

K0ExpJ = Table[ ( K0[(i-101)/20,2] *
                  Exp[ J[[i]] / (I h) ] ),
         {i,1,200} ]
```

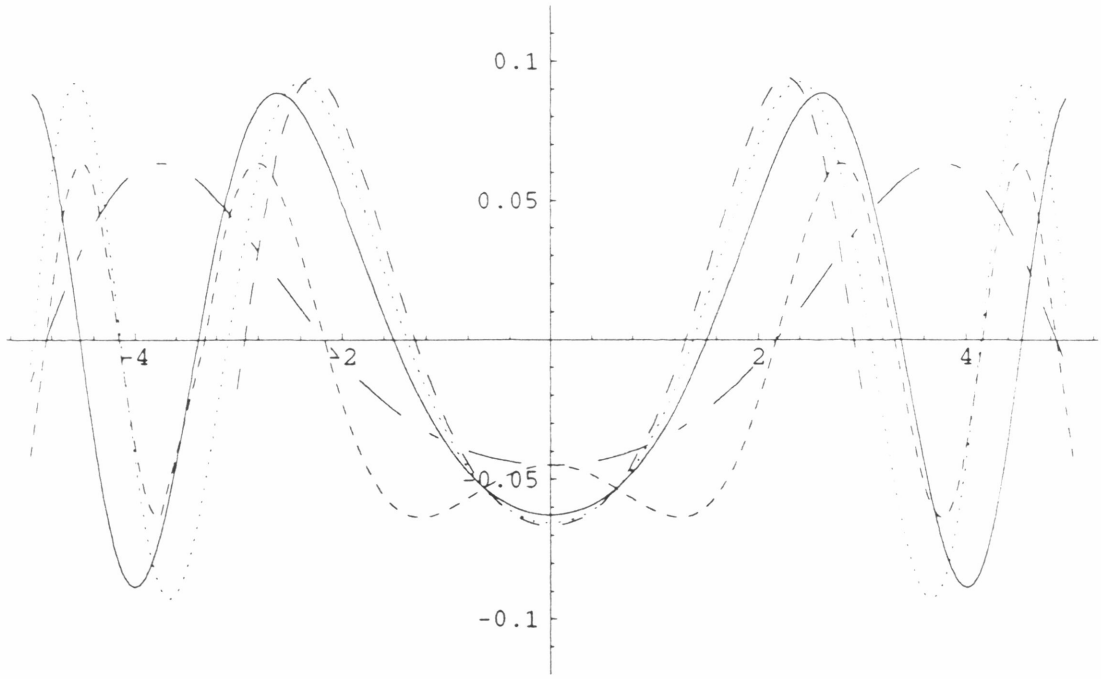
☐ Comparison of the Real parts of K_0, K_0Exp[J], K_0Exp[J]Exp[1/m J1], K_0Exp[J]Exp[1/m J1 + 1/m^2 J2] and the exact propagator

```
Show[Plot[Re[K0[x,t]], {x,-5,5},
        PlotStyle->Dashing[{.01,.01}]],
     ListPlot[Table[{(i-101)/20, Re[K0ExpJ[[i]] ]},
        {i,1,200}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
     ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,1,200}], PlotJoined->True, AxesOrigin->{0,0} ],
     ListPlot[Table[{(i-101)/20,
        Re[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,1,200}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.002,.01}]],
     Plot[Re[Ktrue[x,t]], {x,-5,5},
        PlotStyle->Dashing[{.02,.02}]],
     PlotRange->{-.12,.12} ]
```



☐ Comparison of the Imaginary parts of K_0, K_0Exp[J], K_0Exp[J]Exp[1/m J1], K_0Exp[J]Exp[1/m J1 + 1/m^2 J2] and the exact propagator

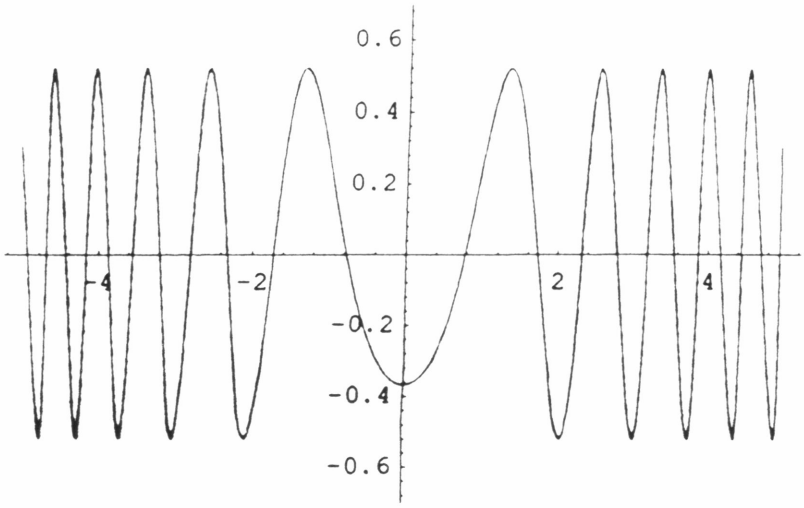
```
Show[Plot[Im[K0[x,t]], {x,-5,5},
        PlotStyle->Dashing[{.01,.01}]],
     ListPlot[Table[{(i-101)/20, Im[K0ExpJ[[i]] ]},
        {i,1,200}], PlotJoined->True, AxesOrigin ->{0,0},
        PlotStyle->Dashing[{.01,.03,.05,.03}]],
     ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] * Exp[1/m J1[[i]]]) ]},
        {i,1,200}], PlotJoined->True, AxesOrigin->{0,0} ],
     ListPlot[Table[{(i-101)/20,
        Im[(K0ExpJ[[i]] *
        Exp[1/m J1[[i]] + 1/m^2 J2[[i]] ]) ]},
        {i,1,200}], PlotJoined->True, AxesOrigin->{0,0},
        PlotStyle->Dashing[{.002,.01}]],
     Plot[Im[Ktrue[x,t]], {x,-5,5},
        PlotStyle->Dashing[{.02,.02}]],
     PlotRange->{-.12,.12} ]
```



1-Dim Inverse vs. Normal Harmonic Osc.
Imaginary Part of y = 0, t = 1.0

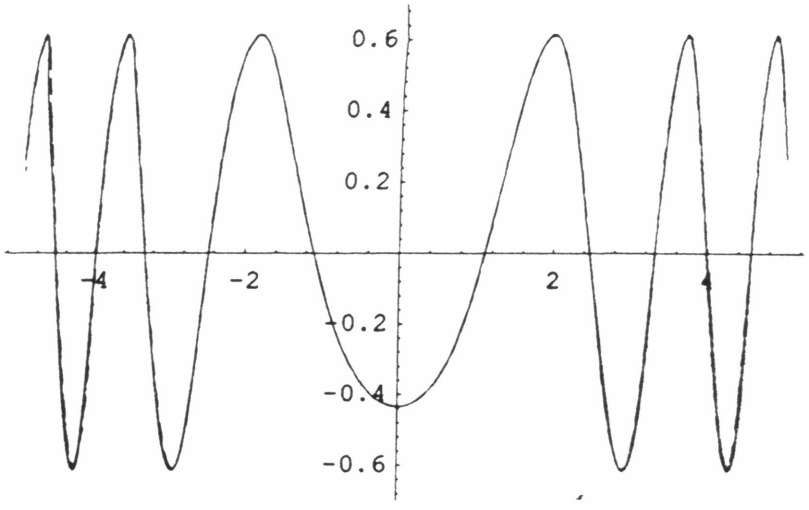
In[322]:=

```
Show[ImCalcPlotJ1,ImCalcPlotJ2,ImTrueInvH,
     PlotRange->{-0.7,0.7}];
```



In[323]:=

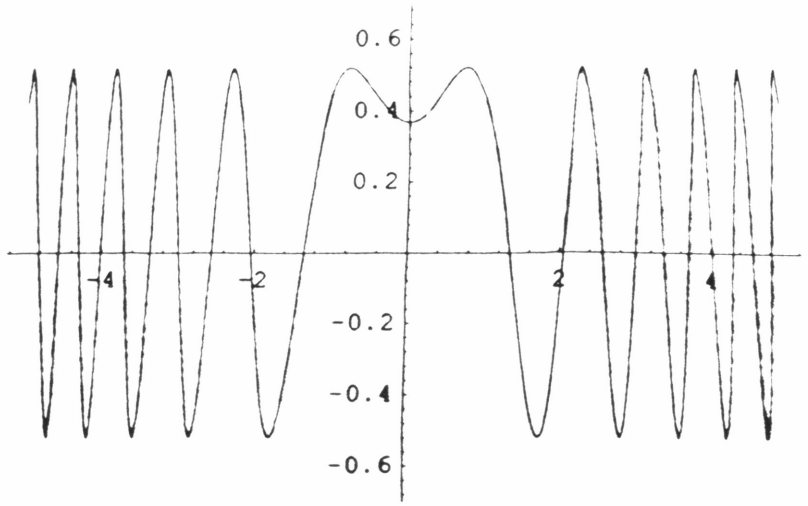
```
Show[ImCalcPlotT1,ImCalcPlotT2,ImTrueH,
     PlotRange->{-0.7,0.7}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 0, t = 1.0

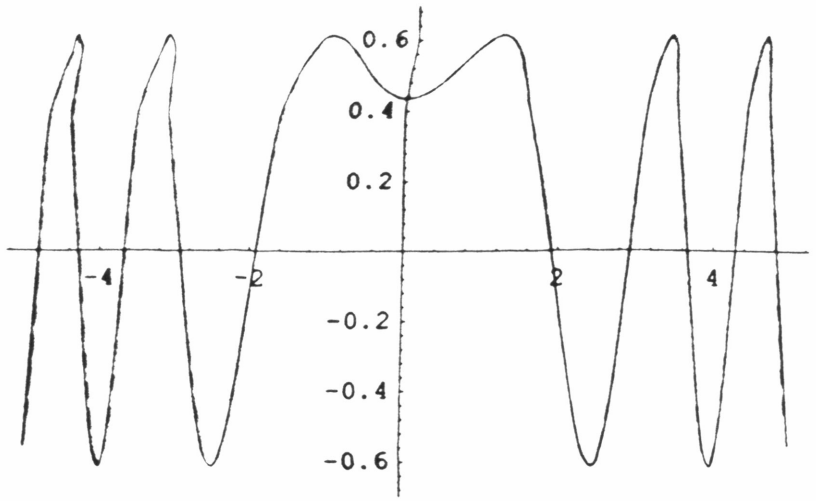
In[318]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-0.7,0.7}];
```



In[319]:=

```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-0.7,0.7}];
```



- - - - - - 1ST ORDER

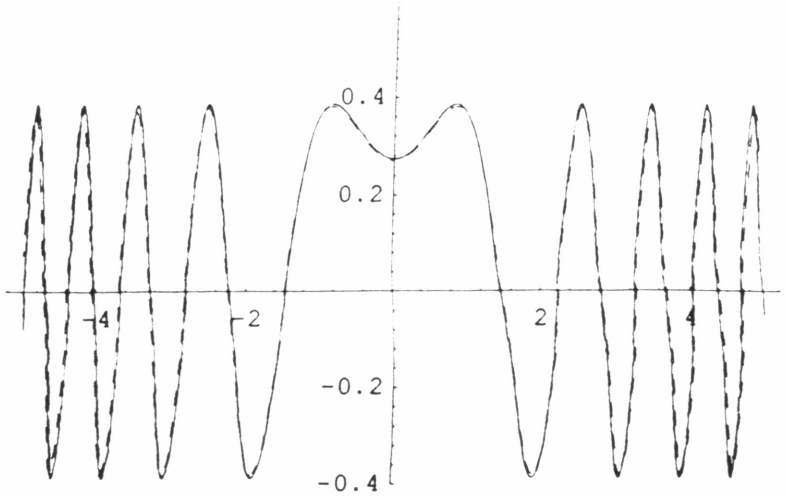
— — - — 2nd ORDER

——————— EXACT

1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 0, t = 1.5

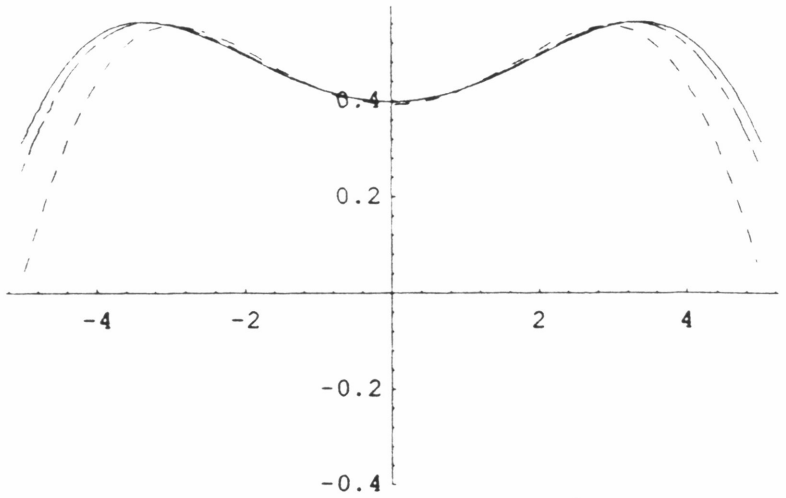
In[132]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-0.4,0.6}];
```



In[131]:=

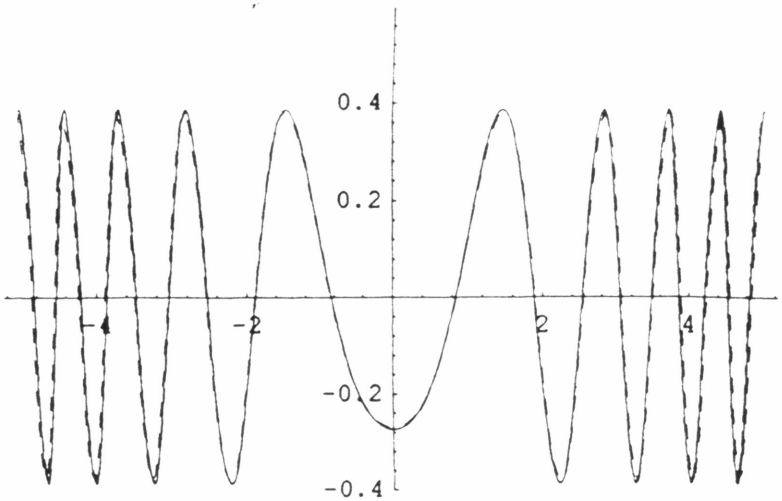
```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-0.4,0.6}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Imaginary Part of y = 0, t = 1.5

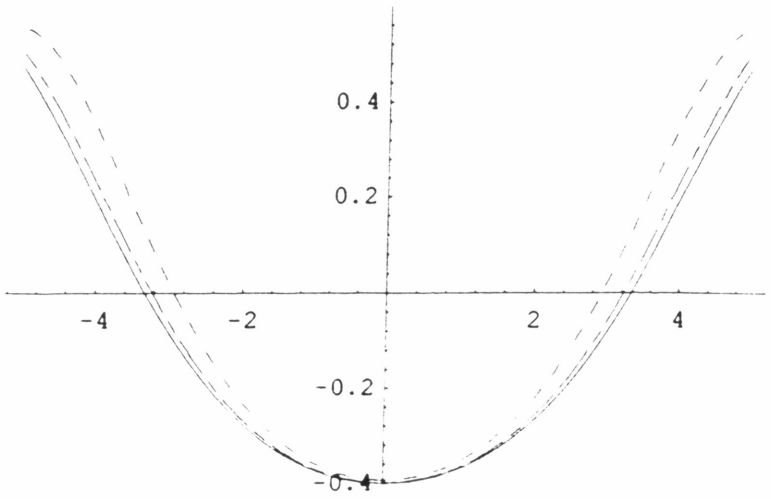
In[130]:=

```
Show[ImCalcPlotJ1,ImCalcPlotJ2,ImTrueInvH,
     PlotRange->{-0.4,0.6}];
```



In[129]:=

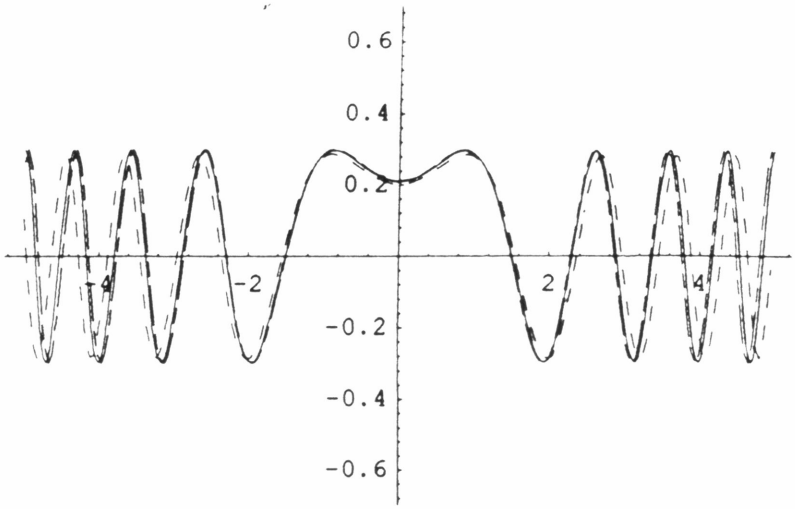
```
Show[ImCalcPlotT1,ImCalcPlotT2,ImTrueH,
     PlotRange->{-0.4,0.6}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 0, t = 2.0

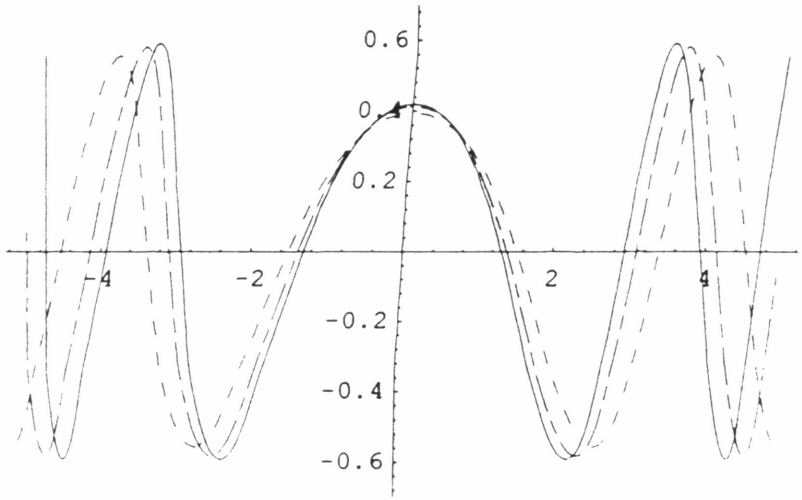
In[357]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-0.7,0.7}];
```



In[358]:=

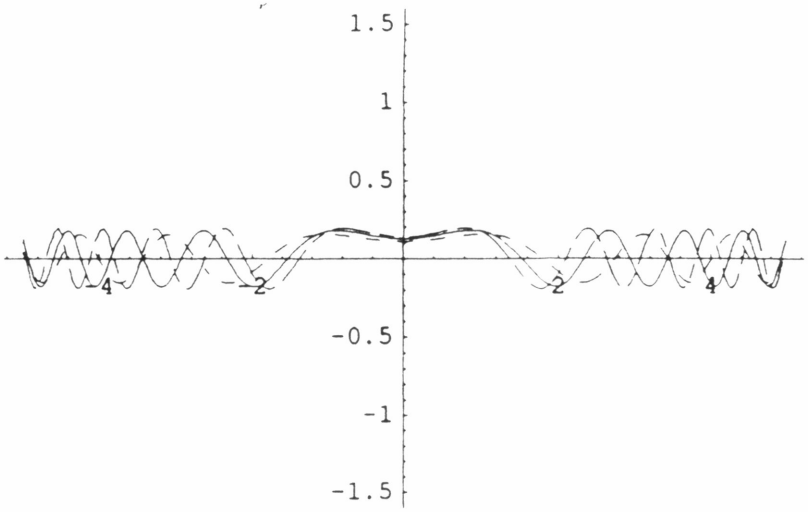
```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-0.7,0.7}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 0, t = 3.0

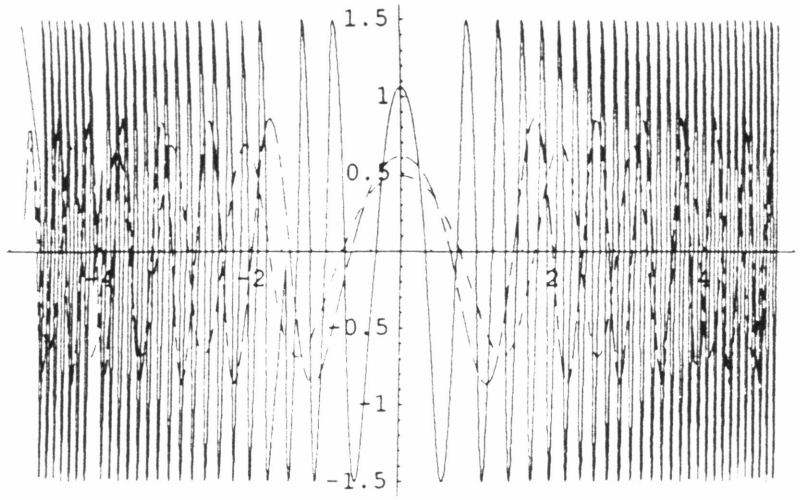
In[33]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-1.6,1.6}];
```



In[34]:=

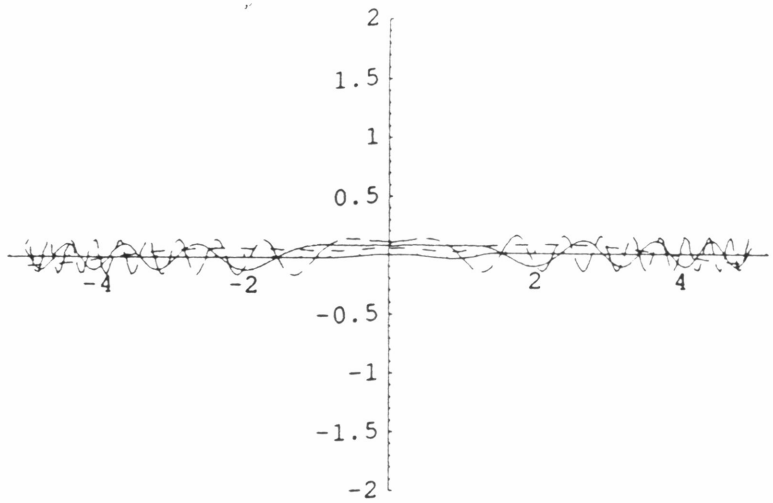
```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-1.6,1.6}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 0, t = 4.0

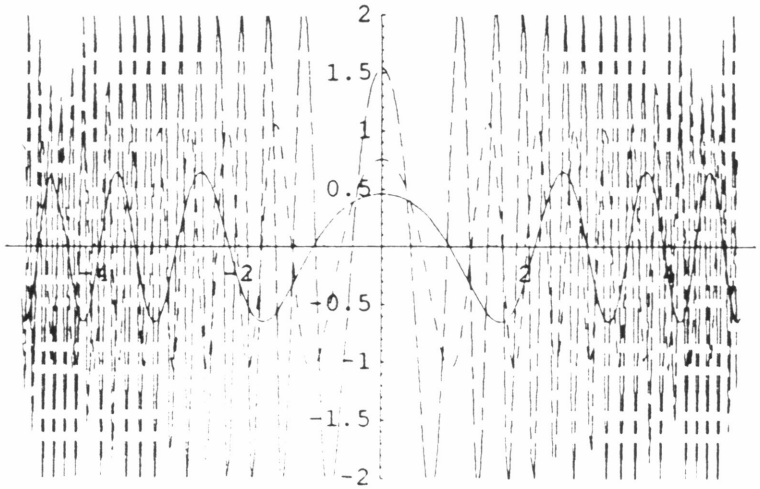
In[34]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-2,2}];
```



In[35]:=

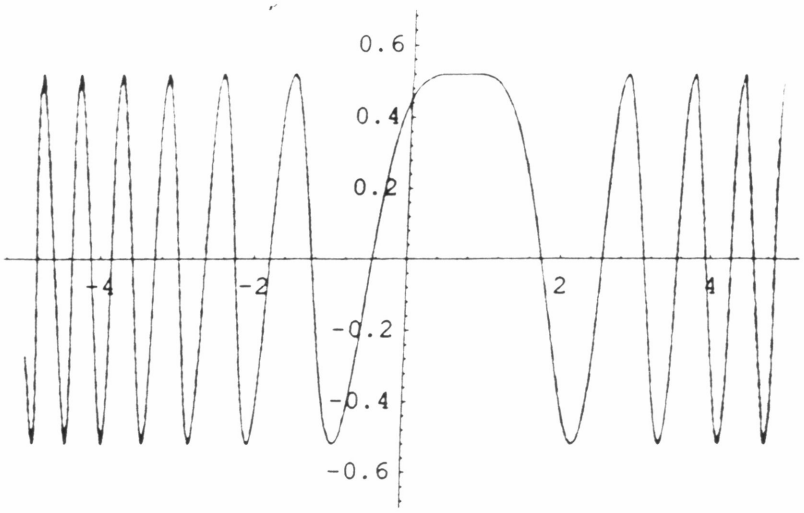
```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-2,2}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 1, t = 1.0

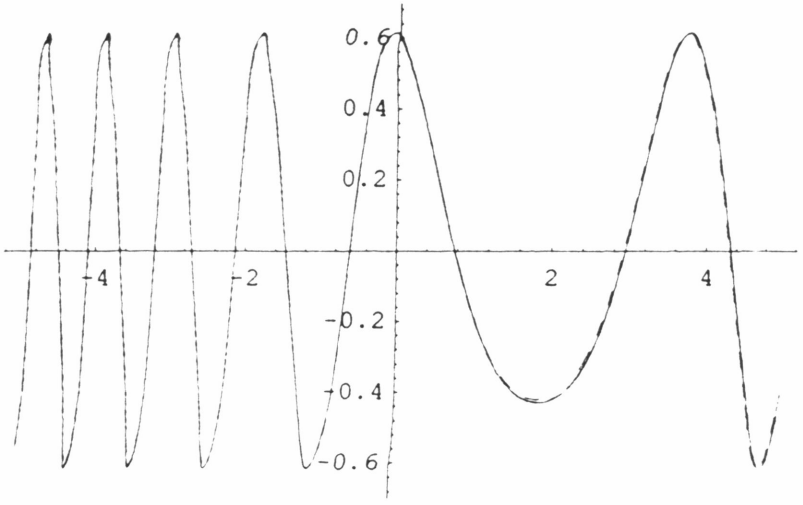
In[55]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-0.7,0.7}];
```



In[54]:=

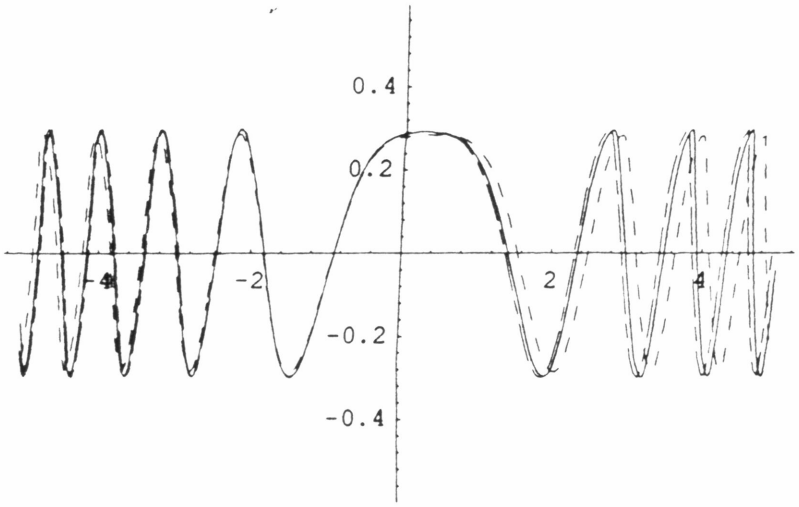
```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-0.7,0.7}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 1, t = 2.0

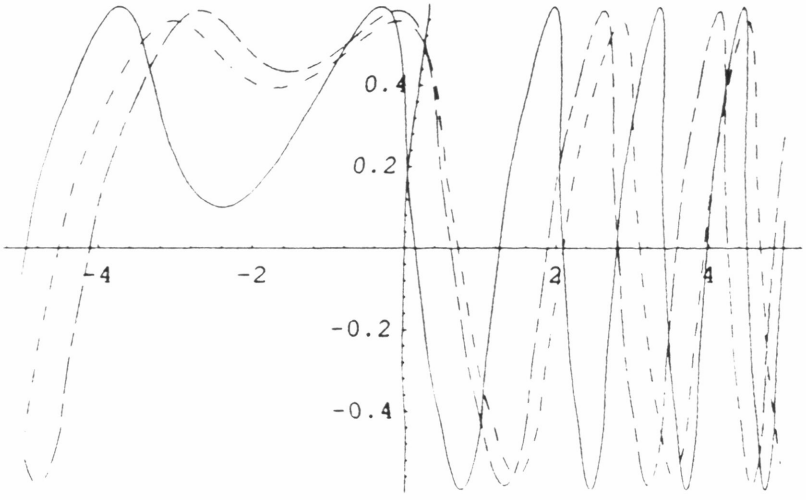
In[198]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-0.6,0.6}];
```



In[199]:=

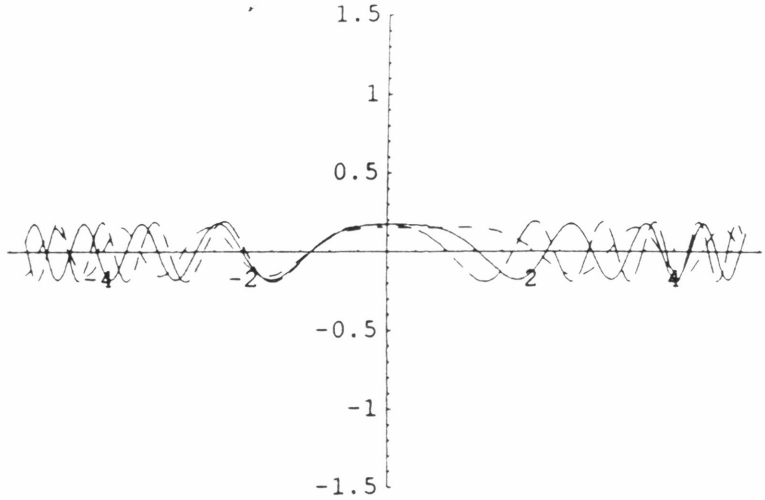
```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-0.6,0.6}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 1, t = 3.0

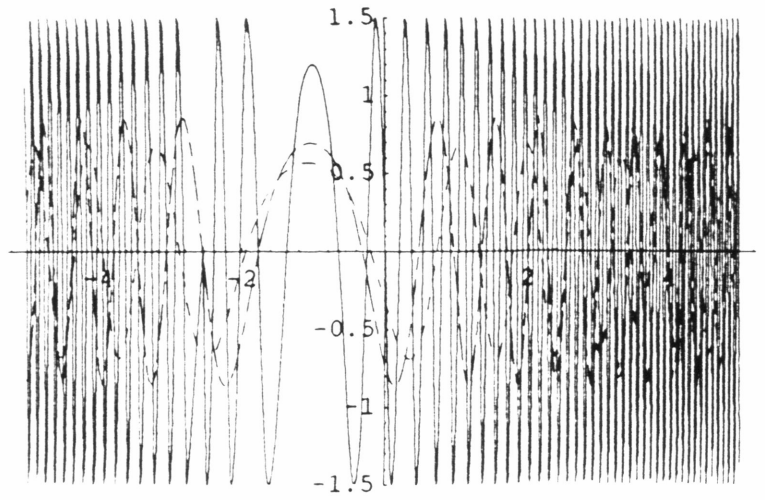
In[245]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-1.5,1.5}];
```



In[242]:=

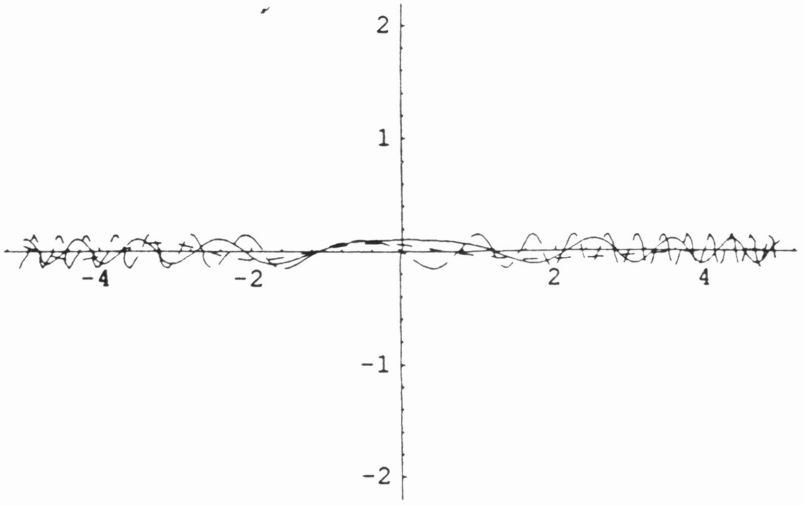
```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-1.5,1.5}];
```



1-Dim Inverse vs. Normal Harmonic Osc.
Real Part of y = 1, t = 4.0

In[279]:=

```
Show[ReCalcPlotJ1,ReCalcPlotJ2,ReTrueInvH,
     PlotRange->{-2.2,2.2}];
```



In[280]:=

```
Show[ReCalcPlotT1,ReCalcPlotT2,ReTrueH,
     PlotRange->{-2.2,2.2}];
```