Contents

October 11, 2002

Chapter 1

Animated Wave Packet–Wave Packet Scattering

Jon J.V. Maestri and Rubin H. Landau, with Kirk Rowe

Abstract A simple and explicit technique for the numerical solution of the time-dependent two-particle Schrödinger equation is developed and applied to wave packet-wave packet scattering in one dimension. The particle-particle potential can be an arbitrary function of the coordinates of the particles, the initial and boundary conditions can be arbitrary, and the techniques can be used for any number of dimensions. The resulting solution is displayed in plots and animations. The lab is supplemented by materials on the web [1].

1.1 Introduction

Quantum mechanics textbooks often present a time-independent view of a single particle interacting with an external potential, rather than the more realistic time-dependent view of two particles interacting with each other. In part, this makes the physics clearer, and in part, this reflects the difficulty of solving the time-independent two–particle Schrödinger equation for the motion of wave packets. In the classic quantum mechanics text by Schiff [3], examples of realistic quantum scattering, such as that on the left in Fig. 1.1, are produced by computer simulations of wave packets colliding with square potential barriers and wells. Generations of students have carried memories of these images (or of the film loops containing these frames [4]) as examples of what realistic quantum scattering looks like.

While the left part of Fig. 1.1 is a good visualization of a quantum scattering processes, this lab extends the simulations of realistic quantum interactions to include particle–particle scattering when both particles are represented by wave packets. Although more complicated, this, presumably, is closer to nature and may illustrate some physics not usually found in quantum mechanics textbooks. In addition, this extension goes beyond the treatment found in most computational physics texts which concentrate on highly restricted forms of *two-particle* wave packets [7], or *one-particle* wave packets [8, 9, 10].

The simulations of the time-dependent Schrödinger equation shown by Schiff were based on the 1967 finite-difference algorithms developed by Goldberg *et al.* [4]. Those simulations, while revealing, had problems with stability and probability conservation. A decade later, Cakmak and Askar [5] solved the stability problem by using a better approximation for the time derivative. After yet another decade, Visscher [6] solved the probability conservation problem by solving for the real and imaginary parts of the wave function at slightly different ("staggered") times.

In this lab we combine some recent advances in the numerical solution of the *two particle*—in contrast to the *one particle* —time-dependent Schrödinger equation. Other

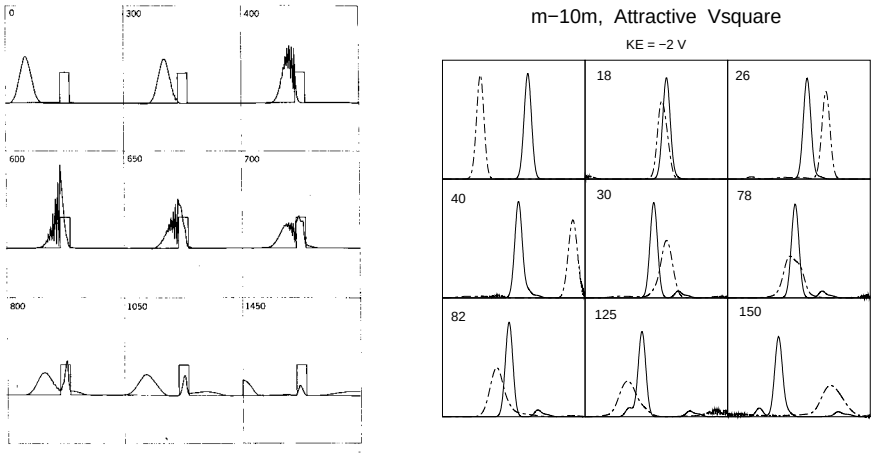

Figure 1.1: *Left:* A time sequence of a Gaussian wave packet scattering from a square barrier with the mean energy equal to the barrier height, as taken from the textbook by Schiff. *Right:* A time sequence of a Gaussian wave packet representing one particle scattering from another Gaussian wave packet representing a different particle.

than being independent of spin, no assumptions are made regarding the functional form of the interaction or initial conditions, and, in particular, there is no requirement of separation into relative and center-of-mass variables.[7] The method is simple, explicit, robust, easy to modify, memory preserving, and useful in research. However, high precision does require small time and space steps, and, consequently, long running times. A similar approach for the time-dependent one-particle Schrödinger equation in a two-dimensional space can be found in the last chapter of the *Computational Physics* text [8].

1.2 Two-Particle Schrödinger Equation

We solve the two-particle time-dependent Schrödinger equation

$$i\frac{\partial}{\partial t}\psi(x_1,x_2,t) = H\psi(x_1,x_2,t), \tag{1.1}$$

$$H = -\frac{1}{2m_1}\frac{\partial^2}{\partial x_1^2} - \frac{1}{2m_2}\frac{\partial^2}{\partial x_2^2} + V(x_1,x_2). \tag{1.2}$$

Here H is the Hamiltonian operator, m_i and x_i are the mass and position of particle $i = 1, 2$, and, for simplicity, we assume a one-dimensional space and set $\hbar = 1$. Knowledge of the two-particle wave function $\psi(x_1, x_2, t)$ permits the calculation of the probability density for particle 1 being at x_1 and particle 2 being at x_2 at time t via the standard definition:

$$\rho(x_1,x_2,t) = |\psi(x_1,x_2,t)|^2 . \tag{1.3}$$

The fact that particles 1 and 2 must be located someplace in space leads to the normalization constraint on the wave function at each instant of time:

$$P(t) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} dx_1\,dx_2\,|\psi(x_1,x_2,t)|^2 = \text{constant}. \tag{1.4}$$

The simplification of the two-body wave function into a single-particle one $\psi(x)$ is an approximation unless the system is uncorrelated (in which case the total wave function

can be written in product form). However, it is possible to deduce meaningful one-particle densities from the two-particle density by integrating over the other particle:

$$\rho_1(x_i,t) = \int_{-\infty}^{+\infty} dx_j\, \rho(x_1,x_2,t), \quad (i \neq j = 1,2). \tag{1.5}$$

Here we use subscripts on the single-particle densities to distinguish them from the two-particle density ρ. Of course, the full solution is $\rho(x_1,x_2,t)$, but as we shall see, it is somewhat of a challenge to see the physics in a three-variable function, and so sometimes view $\rho_1(x_1,t)$ and $\rho_2(x_2,t)$ as two separate wave packets colliding.

If particles 1 and 2 are identical, then their total wave function should be symmetric or antisymmetric under interchange of the particles. We impose this condition on the numerically-determined space wave function $\psi(x_1,x_2)$, by forming the combinations

$$\psi^{'}(x_1,x_2) = \frac{1}{\sqrt{2}}\left[\psi(x_1,x_2) \pm \psi(x_2,x_1)\right] \Rightarrow \tag{1.6}$$

$$\rho(x_1,x_2) = \frac{1}{2}\left|\psi(x_1,x_2)\right|^2 + \frac{1}{2}\left|\psi(x_2,x_1)\right|^2 \pm \mathrm{Re}\left[\psi^*(x_1,x_2)\psi(x_2,x_1)\right]. \tag{1.7}$$

The cross term in (1.7) places a correlation into the wave function in addition to that caused by the particle-particle interaction.

1.3 Numerical Method

We solve the two-particle Schrödinger equation (1.2) via a finite difference method which converts the partial differential equation into a set of simultaneous algebraic equations. First, we evaluate the dependent variable ψ upon a grid of discrete values for the independent variables:[4]

$$\psi(x_1,x_2,t) = \psi(x_1 = l\Delta x_1, x_2 = m\Delta x_2, t = n\Delta t) \;\equiv\; \psi^n_{l,m}, \tag{1.8}$$

where l, m, and n are integers. For each time or n value, we determine the spatial part of the wave function over an entire two-dimensional grid of x_1 and x_2 values. The time is then advanced by one step, and the solution over the spatial grid is found again.

The space part of the algorithm is based on Taylor expansions of $\psi(x_1,x_2,t)$ in *both* the x_1 and x_2 variables up to $\mathcal{O}(\Delta x^4)$; for example,

$$\frac{\partial^2\psi}{\partial x_1^2} \simeq \frac{\psi(x_1+\Delta x_1,x_2) - 2\psi(x_1,x_2) + \;\psi(x_1-\Delta x_1,x_2)}{\Delta x_1^2} + \mathcal{O}(\Delta x_1^2). \tag{1.9}$$

In discrete notation, the RHS of the Schrödinger equation (1.2) now becomes:

$$H\psi = -\frac{\psi_{l+1,m} - 2\psi_{l,m} + \psi_{l-1,m}}{2m_1\Delta x_1^2} - \frac{\psi_{l,m+1} - 2\psi_{l,m} + \psi_{l,m-1}}{2m_2\Delta x_2^2} + V_{lm}\psi_{l,m}. \tag{1.10}$$

Next, we express the time derivative in (1.1) in terms of finite time differences by taking the formal solution to the time-dependent Schrödinger equation and making a forward-difference approximation for time evolution operator:

$$\psi^{n+1}_{l,m} = e^{-i\Delta t H}\psi^n_{l,m} \simeq (1 - i\Delta t H)\psi^n_{l,m}. \tag{1.11}$$

While simple, this particular approximation scheme is unstable since The term multiplying ψ has eigenvalue $(1 - iE\Delta t)$ and so a growing modulus $\sqrt{1+E^2\Delta t^2}$ which increases with each time step [5] [9]. The improvement developed by Askar and Cakmak [5] is a central

difference algorithm also based on the formal solution (1.11):

$$\psi_{l,m}^{n+1} - \psi_{l,m}^{n-1} = \left(e^{-i\Delta t H} - e^{i\Delta t H}\right)\psi_{l,m}^{n} \simeq -2i\Delta t H \psi_{l,m}^{n}, \tag{1.12}$$

$$\Rightarrow \quad \psi_{l,m}^{n+1} \simeq \psi_{l,m}^{n-1} - 2i\left[\left\{(\frac{1}{m_1}+\frac{1}{m_2})4\lambda + \Delta x V_{l,m}\right\}\psi_{l,m}^{n} - \lambda\left\{\frac{1}{m_1}(\psi_{l+l,m}^{n}+\psi_{l-1,m}^{n}) + \frac{1}{m_2}(\psi_{l,m+1}^{n}+\psi_{l,m-1}^{n})\right\}\right], \tag{1.13}$$

where we have assumed $\Delta x_1 = \Delta x_2$ and formed the ratio

$$\lambda = \Delta t/\Delta x^2. \tag{1.14}$$

Equation (1.13) is an *explicit* solution in which the wave function for one future time is determined from the wave function of two past times. All future times are then generated by continued iteration, with the wave function at only two time values needed to be stored simultaneously in memory. In contrast, an *implicit* solution determines the wave function for all future times in just one step, yet this one step requires the solution of simultaneous algebraic equations involving all space and time values. Accordingly, an implicit solution may be more direct and quicker, yet it requires the inversion of exceedingly large ($\sim 10^{10} \times 10^{10}$) matrices.

Although the explicit method outlined above produces a solution which is stable and second-order accurate in time, in practice it does not conserve probability very well. For scattering problems, Visscher[6] has deduced an improvement which takes advantage of the extra degree of freedom provided by the complexity of the wave function to preserve probability better. If we separate the wave function into real and imaginary parts,

$$\psi_{l,m}^{n+1} = u_{l,m}^{n+1} + i\, v_{l,m}^{n+1}, \tag{1.15}$$

the algorithm (1.13) separates into the pair of coupled equations:

$$u_{l,m}^{n+1} = u_{l,m}^{n-1} + 2\left[\left\{(\frac{1}{m_1}+\frac{1}{m_2})4\lambda + \Delta t V_{l,m}\right\}v_{l,m}^{n} - \lambda\left\{\frac{1}{m_1}(v_{l+1,m}^{n}+v_{l-1,m}^{n}) + \frac{1}{m_2}(v_{l,m+1}^{n}+v_{l,m-1}^{n})\right\}\right], \tag{1.16}$$

$$v_{l,m}^{n+1} = v_{l,m}^{n-1} - 2\left[\left\{(\frac{1}{m_1}+\frac{1}{m_2})4\lambda + \Delta t V_{l,m}\right\}u_{l,m}^{n} - \lambda\left\{\frac{1}{m_1}(u_{l+1,m}^{n}+u_{l-1,m}^{n})\frac{1}{m_2}(u_{l,m+1}^{n}+u_{l,m-1}^{n})\right\}\right]. \tag{1.17}$$

Visscher's advance evaluates the real and imaginary parts of the wave function at slightly different (staggered) times,

$$[u_{l,m}^{n},\, v_{l,m}^{n}] = [\mathrm{Re}\psi(x,t),\, \mathrm{Im}\psi(x,t+\frac{1}{2}\Delta t)], \tag{1.18}$$

and uses a definition for probability density that differs for integer and half-integer time steps,

$$\rho(x,t) = |\mathrm{Re}\psi(x,t)|^2 + \mathrm{Im}\psi(x,t+\frac{\Delta t}{2})\,\mathrm{Im}\psi(x,t-\frac{\Delta t}{2}), \tag{1.19}$$

$$\rho(x,t+\frac{\Delta t}{2}) = \mathrm{Re}\psi(x,t+\Delta t)\mathrm{Re}\psi(x,t) + \left|\mathrm{Im}\psi(x,t+\frac{\Delta t}{2})\right|^2. \tag{1.20}$$

These definitions reduce to the standard one for infinitesimal Δt, and provide an algebraic cancelation of errors so that probability is conserved.

Table 1: antisymmetrized m-m collision via attractive square well

Parameter	Value	Interpretation
Δx	0.001	space step
Δt	2.5×10^{-7}	time step
k_1	$+157$	particle 1 momentum
k_2	-157	particle 2 momentum
σ	0.05	wave packet width
x_1^0	467 (0.33×1401 steps)	initial center of 1
x_2^0	934 (0.667×1401 steps)	initial center of 2
$N_1 = N_2$	1399	interior number of space steps
L	1.401 (1401 space steps)	box size
T	0.005 (20,000 time steps)	total time
V_0	$-100,000$	potential depth
α	0.062	potential size

1.4 Simulation

We assume that the particle–particle potential is central and depends only on the relative distance between particles 1 and 2 (the method can handle any x_1 and x_2 functional dependences). As a simple case, we assume a square-well dependence of range α and depth V_0:

$$V(x_1, x_2) = V_0 \; \theta(\alpha - |x_1 - x_2|). \tag{1.21}$$

1.4.1 Initial and Boundary Conditions

We model a scattering experiment in which particle 1, initially at x_1^0 with momentum k_1, collides with particle 2, initially far away at x_2^0 with momentum k_2, by assuming a product of independent wave packets for particles 1 and 2:

$$\psi(x_1, x_2, t = 0) = e^{ik_1 x_1} \exp[-\frac{(x_1 - x_1^0)^2}{4\sigma^2}] \times e^{ik_2 x_2} \exp[-\frac{(x_2 - x_2^0)^2}{4\sigma^2}]. \tag{1.22}$$

The presence of the Gaussian factors in (1.22) produces a state than is not an eigenstate of the particle j momentum operators $-i\partial/\partial x_j$, but instead contains a spread of momenta about the mean, initial momenta k_1 and k_2. Yet if the wave packet is made very broad ($\sigma \to \infty$), we would obtain momentum eigenstates.

Note, that while the Schrödinger equation may separate into one equation in the relative coordinate x and another in the center-of-mass coordinate X, the initial condition (1.22) cannot be written as separate conditions on x and X. Accordingly, a solution of the equation for each particle coordinate is required [7].

We start the staggered-time algorithm with the real part the wave function (1.22) at $t = 0$ and the imaginary part at $t = \Delta t/2$. The initial imaginary part follows by assuming that $\Delta t/2$ is small enough, and σ is large enough, for the initial time dependence of the wave packet to be that of the plane wave parts:

$$\begin{aligned} \mathrm{Im}\psi(x_1, x_2, t = \frac{\Delta t}{2}) &= \sin\left[k_1 x_1 + k_2 x_2 - \left(\frac{k_1^2}{2m_1} + \frac{k_2^2}{2m_2}\right)\frac{\Delta t}{2}\right] \\ &\quad \times \exp - \left[\frac{(x_1 - x_1^0)^2 + (x_2 - x_2^0)^2}{2\sigma^2}\right]. \end{aligned} \tag{1.23}$$

In a scattering experiment, the projectile enters from infinity and the scattered particles are observed at infinity. We model that by solving our partial differential equation within a box of side L that should be much larger than both the range of the potential and the width of the initial wave packet. This leads to the boundary conditions

$$\psi(0, x_2, t) = \psi(x_1, 0, t) = \psi(L, x_2, t) = \psi(x_1, L, t) = 0. \tag{1.24}$$

The largeness of the box minimizes the effects of the boundary conditions during the collision of the wave packets. While we shall see some interesting physics when the wave packets hit the box, this is not related to what happens in an actual scattering experiment.

Some typical parameters used in our tests are given in the Table 1. These parameters were chosen to be similar to those used in the one-particle studies by Goldberg *et al.*. Our space step size $\Delta x = 0.001$ is 1/1000th of the size of the box L, and 1/70th of the size of the wave packet ($\sqrt{2}\sigma \simeq 0.07$). Our time step $\Delta t = 2.5 \times 10^{-7}$ is 1/20,000th of the total time T, and 1/2000th of a typical time for the wave packet ($2\pi/(k_1^2/2m_1) \simeq 5 \times 10^{-4}$). This small a time step was chosen to obtain a high degree of probability conservation. A larger step size may be used if your main interest is in viewing the physics.

The combination of time step and space step sizes were determined by trial and error until values were found which provided stability and precision (too large a Δx leads to spurious ripples during interactions). In general, stability is obtained by making Δt small enough [6], with a simultaneous changes made in Δt and Δx to keep constant the value of

$$\lambda = \frac{\Delta t}{\Delta x^2}. \tag{1.25}$$

Total probability, as determined by a double Simpson's-rule integration in (1.4), is typically conserved to 13 decimal places, impressively close to machine precision. In contrast, the mean energy, for which we do not use a definition optimized for staggered time, is conserved only to 3 decimal places.

1.4.2 Running the Simulation

As a sample of the results showing the two-particle particle density as a function of both coordinates for three times, in Fig. 1.2 we show the two-particle density $\rho(x_1, x_2, t)$ as a function of the particle positions x_1 and x_2. This is for an $m - 10m$ collision in which the heavy particle at x_1 hardly moves. The different parts correspond to times 0, $T/4$, and $T/2$, where T is the total interaction time. We see that the x_1 position of the peak of $\rho(x_1, x_2, t)$ changes very little with time, which means that the heavy particle does not move much during the collision. In contrast, we see that the x_2 peak describing the light particle increases up to the collision (middle part), and then continues to increase as particle 2 passes through particle 1. We also note that the two particle wave packet becomes broader in the x_2 variable as a consequence of the collision, but that there is little change in the x_1 behavior.

As a sample of the results showing the single particle densities superimposed, in Fig. 1.1 we show nine frames from the movie of an attractive m-m collision in which the mean energy equals one quarter of the well depth. The initial packets are seen to speed up as they approach each other, and at time 60, the centers have already passed through each other. After that, a transmitted and reflected wave for each packet is seen to develop (times 66-78). Finally, each packet appears to capture or pick up a part of the other packets and move off with it (times 110-180).

1.4.3 Codes

We provide you with the C code `square.c` to run. It contains a square-well potential interaction between the particles and has been trimmed in order to run within limited reasonable time and memory constraints. The more complete program `packets.c` contains a Gaussian potential and has additional features that require significant resources to run. The relation between the complex wave function in math notation and that in the program is given by the table below.

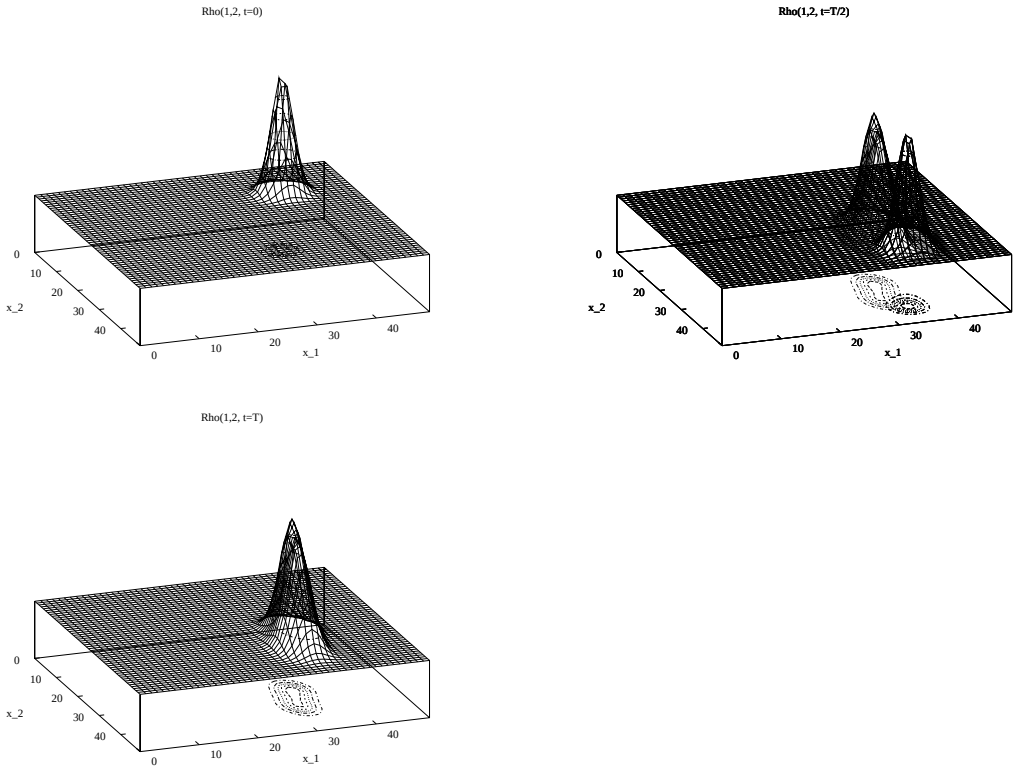

Figure 1.2: The two-particle density $\rho(x_1, x_2)$ as a function of the particle positions x_1 and x_2 for an attractive m-$10m$ collision. The different figures correspond to times 0, $T/2$ (during collision), and T (the end of collision). The heavier particle 1 hardly moves during the collision, while particle 2 moves and broadens.

x_1, x_2, t	l Δx, m Δx, n Δt		
Re$\psi(x_1, x_2, t)$	$u^n_{l,m}$	`RePsi[l][m][n]`	`RePsi[401][401][2]`
Im$\psi(x_1, x_2, t)$	$v^n_{l,m}$	`ImPsi[l][m][n]`	`ImPsi[401][401][2]`
`RePsi[l][m][0]`	Re$\psi(l\Delta x, m\Delta x, t)$	past	
`RePsi[l][m][1]`	Re$\psi(l\Delta x, m\Delta x, t + \Delta t)$	present	
`ImPsi[l][m][0]`	Im$\psi(l\Delta x, m\Delta x, , t + \frac{1}{2}\Delta t)$	split present	
`ImPsi[l][m][1]`	Im$\psi(l\Delta x, m\Delta x, t + \frac{3}{2}\Delta t)$	split future	

1.4.4 Internal Control Parameters

The variable `xDim` is the number of space intervals within our one-dimensional box of size L (1.24). The integration algorithm requires this number to be odd. The variable `choice` gives you a choice of output:

Effect of internal `choice` parameter

value	effect
0	output joint probability density $\rho(x_1, x_2)$
1	output two, single-particle densities $\rho(x_1)$ and $\rho(x_2)$ (default)
2	output correlation function
3	output product of single particle densities

1.4.5 Read-In Control Parameters

Table 1 gave some typical parameters for a high precision research run. The text file `in.dat` contains the majority of control parameters supplied with the program. The file must exist in the same directory as the executable. A typical `in.dat` file is the second column of numbers in table below

Control parameters in file input file `in.dat`

Parameter	Value	Definition
`Nt`	`2000`	number of time steps
`k1`	`110.`	momentum magnitude of particle 1
`k2`	`110.`	momentum magnitude of particle 2 (-x direction)
`dx`	`0.005`	space step size $\Delta x_1 = \Delta x_2 = \Delta x = 1/x_{DIM}$
`dt`	`4.2E-06`	time step Δt
`sig`	`0.05`	wave packet width parameter σ
`x01`	`0.25`	location of initial center of particle 1's wave packet x_1^0
`x02`	`0.60`	location of initial center of particle 2's wave packet x_2^0
`m1`	`0.5`	particle 1's mass m_1
`m2`	`5.`	particle 2's mass m_2
`vmax`	`+90000.`	potential depth V_0 (negative for attractive potential)
`alpha`	`0.062`	potential width parameter α
`nprint`	`100`	print out data for every `nprint` time steps
`symmetry`	`0`	Flag for wave function symmetry -1 antisymmetrized 0 no symmetrization +1 symmetrized

Note that if the time step is made too large, then the algorithm clearly becomes inaccurate, and actually unstable. A detailed analysis of the numerics indicates that instability sets when when the product of the time step and potential exceeds approximately 0.38

$$\Delta t V_0 \geq 0.38 \tag{1.26}$$

1.4.6 Running

Within a shell, you can compile the code with the usual command:

`cc -lm square.c -o square`	Compile code

The `-lm` option permits access to the math library and the argument of the `-o` option `square` is the name of the file into which the executable will be stored. You may also use `gcc` for the *gnu* compiler or a development studio. Additional information about the compiler and its options might be obtained after issuing some variant of the commands:

`man cc man gcc`	read the manual pages
`cc -flags`	tell me about the options

The code is run by entering the name of the executable:

`man cc, man gcc`	read the manual pages
`square`	Run square

1.4.7 Output Files

As the program runs, a series of data files are placed in the directory in which the code is running. We recommend that you keep each run, and the data files that it produces, in a separate subdirectory. This may keep you from intermixing and confusing several runs, and let you go back to collect the data again if some processing fails. For the light program `square.c` with `choice` = 1, the output files are

Output File	Contents
in.dat	*Your* input data file
E_t.dat	total energy E (double space integral of Hamiltonian) *vs* t.
logE_t.dat	Relative energy change, $\log_{10} \frac{E(t)-E(0)}{E(0)}$
logP_t.dat	Relative probability change, $\log_{10} \frac{P(t)-P(0)}{P(0)}$
params.dat	printout of names and values of parameters
run.ti	x, $\rho_1(x)$, $\rho_2(x)$, at time `ti`
SumRho.dat	$\rho_1(x) + \rho_2(x)$ in rows for a surface plot
V.dat	$V(x_1 - x_2)$ in rows for a surface plot

1.5 Tuning and Exploration

1.5.1 Time Steps

As stated above, a detailed analysis of our integration algorithm finds it to become unstable if the product of the time step and the potential exceeds approximately 0.38

$$\Delta t V_0 \geq 0.38 \tag{1.27}$$

The instability shows itself by an increasing magnitude of the wave function.

Verify this assertion by running `square` with successively increasing time steps such that the product of 0.38 is gradually exceeded. As the time steps are increased, `Nt` and `nprint` should also be adjusted so that the total time duration of the scattering process (`Nt*dt`) and the number of output files is approximately the same.

As the product 0.38 is exceeded, the area under the packets should increase both with subsequent times, or at a fixed time as Δt increases. Eventually you may run into overflow.

What might you expect if the time step was just barely stable (a product close to 0.38) and the code allowed to run for a long time duration?

1.5.2 Space Steps

Making the space step Δx too large also leads to inaccuracies. Progressively increase the space step and look at the signs of the inaccuracy. Since the highest level of precision is needed when the wave function changes most rapidly, that is, during the interaction of the two packets, you should notice spurious ripples occurring then. Early publications saw these ripples and have believed them to be interesting new physics; we see here that they are just signs of computational inaccuracy.

1.5.3 Probability Conservation

One of the output files produced by running the code is `logP_t.dat`. It gives $\log_{10} \frac{P(t)-P(0)}{P(0)}$ as a function of time t, where $P(t)$ is the two-particle probability density integrated (with Simpson's rule) over all space,

$$P(t) = \int_{-\infty}^{+\infty} dx_1 \int_{-\infty}^{+\infty} dx_2 \left|\psi(x_1, x_2)\right|^2 . \tag{1.28}$$

Since $\frac{P(t)-P(0)}{P(0)}$ is the relative error in the integrated probability, the $\log_{10}$ of it is essentially the number of places of precision obtained in the probability. In the original paper[2], the space and time sizes were adjusted until the total probability was conserved to 13 decimal places (impressively close to machine precision).

Plot up the data in `logP_t.dat` to see what level of precision your runs are obtaining. Now try to adjust the time and space step to get 13 decimal places. Note, as indicated in the theory section, analysis[6] predicts that stability requires

$$\lambda = \Delta t/\Delta x^2 \tag{1.29}$$

to be kept approximately constant. This means that we cannot make one smaller without also making the other smaller, but in proportion!

1.5.4 Wave function Plots

As the program runs progressively through the time loop, after every `nprint` values of time a file with the name `run.ti` will be outputted, where `ti` is the time. For example, with `nprint`=100) we would obtain

`run.00001, run.00100, run.00200, ...`

For `choice`=1, each of these data files will consist of three columns with the values of x, $\rho_1(x)$, and $\rho_2(x)$. Use your favorite 2D plotting package to make plots of these densities for a number of interesting times, namely, before, during, and after collision, as well bouncing off the wall. Overlay both densities on the same graph, using solid and colored dashed lines to distinguish the two. Your plots should look like those on the right of Fig. 1.1.

We recommend that you make these plots with the popular package *gnuplot*, since we will next use *gnuplot*'s output to make movies. *gnuplot* is free and is available for both Unix and Windows. On Unix, you can get information on running it with the

```
> man gnuplot                    List manual pages for gnuplot
```

,

or in the Landau-Fink book. Other plotting packages (such as Canvas and Illustrator) that do 2D plots and then lets you paste them together for animations would also be fine. Within *gnuplot* the densities for one time and all space are plotted with the subcommand:

```
plot 'run.00100' using 1:2 w l, plot 'run.00100'using 1:3 w l
```

gnuplot command

Investigate

1.5.5 Wave function Movies

In the system described below, movies are made by creating a sequence of plots (frames) for different times, and then pasting the frame together into an animated gif. When a browser opens an animated gif file, it automatically displays the various frames in a rapid enough sequence for you mind's eye to see the frames as an animation of a continuous event.

Gnuplot itself cannot produce output in the *.gif* format, so we outputted our results into an intermediate *pixmap* format and then used other free program to produce the *gif* images. Since there may be hundreds of frames that need to be produced to make a single movie, with a whole series of command needed for each frame, this would be a tedious task to do by hand. To this end we wrote a script `MakeGifs.script` that automates the task. A *script* is a file containing a series of commands that might otherwise be given to execute within a shell. Once in a script, they can now be executed as a single command, and even looped over via built-in control structures.

MakeGifs.scrip

```
#! /bin/ksh

unalias rm
integer i=$1
while test i -lt $2
do

 if test i -lt 10
  then
  print "set terminal pbm small color; set output\"$3t=0000$i.ppm\"; set noxtics; set noytics;
  set size 1.0, 1.0; set yrange [0:.1];
  plot 'run.0000$i' using 1:2 w lines, 'run.0000$i' using 1:3 w lines;
  " >data0000$i.gnu
  gnuplot data0000$i.gnu
  ppmquant -map samp_colormap $3t=0000$i.ppm>$3at=0000$i.ppm
  ppmtogif -map samp_colormap $3at=0000$i.ppm > $30000$i.gif
  rm $3t=0000$i.ppm $3at=0000$i.ppm data0000$i.gnu
  i=i+99
  fi

if test i -gt 9 -a i -lt 1000
then
print "set terminal pbm small color; set output\"$3t=00$i.ppm\"; set noxtics; set noytics;
set size 1.0, 1.0; set yrange [0:.1];
plot 'run.00$i' using 1:2 w lines, 'run.00$i' using 1:3 w lines;
" >data00$i.gnu
gnuplot data00$i.gnu
ppmquant -map samp_colormap $3t=00$i.ppm>$3at=00$i.ppm
ppmtogif -map samp_colormap $3at=00$i.ppm > $300$i.gif
rm $3t=00$i.ppm $3at=00$i.ppm data00$i.gnu
i=i+100
fi

if test i -gt 999 -a i -lt 10000
then
print "set terminal pbm small color; set output\"$3t=0$i.ppm\"; set noxtics; set noytics;
set size 1.0, 1.0; set yrange [0:.1];
plot 'run.0$i' using 1:2 w lines, 'run.0$i' using 1:3 w lines;
" >data0$i.gnu
gnuplot data0$i.gnu
ppmquant -map samp_colormap $3t=0$i.ppm>$3at=0$i.ppm
ppmtogif -map samp_colormap $3at=0$i.ppm > $30$i.gif
```

```
rm $3t=0$i.ppm $3at=0$i.ppm data0$i.gnu
i=i+100
fi

if test i -gt 9999 -a i -lt 60001
then
print "set terminal pbm small color; set output\"$3t=$i.ppm\"; set noxtics; set noytics;
set size 1.0, 1.0; set yrange [0:.1];
plot 'run.$i' using 1:2 w lines, 'run.$i' using 1:3 w lines;
" >data$i.gnu
gnuplot data$i.gnu
ppmquant -map samp_colormap $3t=$i.ppm>$3at=$i.ppm
ppmtogif -map samp_colormap $3at=$i.ppm > $3$i.gif
rm $3t=$i.ppm $3at=$i.ppm data$i.gnu
i=i+100
fi

done
```

- The script file, along with the file `samp`, should be placed in the directory containing the `run.lmn` output files.
- The #! line tells the computer that the subsequent commands are in the korn shell.
- The symbol $\$i$ indicates that i is an argument to the script. That is, the script is run from a shell by giving the name of the script with three arguments, the beginning time, the time `Ntimes` +1, and the name of the file where you wish your gifs to be stored. For example,

```
> mscript 1 101 OutFile                    make gif from times 1 to 101 in OutFile
```

- The symbol > indicates that the output is directed to the following file.
- The `ppmquant` command takes a pixmap and maps an exsisting set of colors to new ones. For this to work, you must have the supplied map file `samp` in the working directory.
- The counter increments such as `i=i+99` at the bottom of the loops should be adjusted by the user to coincide with the choice of `nprint`. The first counter should increment by the value `nprint-1`, because the first file written is `.00001` not `.00000`.), and the rest of the files by `nprint`.

Depending on the number of frames, it may take 10 minutes or so to run this script. Upon completion of the script, there should be a new set of files of the form `OutfileTime.gif`. Here `Outfile` is your chosen name, `Time` is the familiar `nprint` multiples, and `.gif` indicates that these are gif files. Note that you can examine any or all of these gif files with a Web browser.

The final act of movie making is to merge the individual gif files into an animated gif with the program `gifmerge` (or another one that you may have available). The movie will also be a gif file with the `.gof` extension. It too can be opened with a browser, and that will automatically cause the browser to flip through the various frames. A typical use of `gifmerge` is

```
> gifmerge -10 *.gif > movie                merge all .gif files into movie
```

Here the `-10` separates the frames by 0.1 sec. in actual time, and the * is a wildcard that will include all ġif files in the present directory. Since we constructed these `.gif` files with sequential numbering, `gifmerge` will paste them together in the proper sequence and place them in the file `movie`, that can then be viewed with a browser.

1.5.6 Effect of Symmetrization

Compare movies of the scattering of the wave packet for two equal mass particles when the wave function has been symmetrized and antisymmetrized. It is said that symmetrization is somewhat equivalent to including an additional attractive force between the particle, while antisymmetrization is equivalent to adding a repulsive force or one that causes to

the particles to exchange positions with each other. Describe the effects you see in your simulations and see if they can be viewed in terms of attractions and repulsions.

1.5.7 Effect of the Box

Although fascinating to watch the wave packet decompose itself as it collides with the walls of the box, the collisions with the walls of the box is not something that would occur in a realistic scattering experiment. However, we need to have the box present to solve the problem both in a mathematical and numerical sense. Try out different sizes for the box and see if you can notice any changes in the collision.

1.5.8 Potential Effects

We have chosen a square-well potential for simplicity and to make the interaction abrupt enough to get some rapid changes in the wave packets. Investigate the effects of

1. Changing the potential from attractive to repulsive.
2. Changing the depth of the attractive potential so that the two particles get hold each other together for as long a time as possible (quasi bound states).
3. Making the potential "softer", that is, less abrupt.

Harmonic Oscillator Potential

Investigate the effect of making the potential a harmonic oscillator,

$$V = -\frac{1}{2}k|x_1 - x_2|^2. \tag{1.30}$$

Note that this is a special (and unrealistic) case for which the theory may not be appropriate. Since this potential gets stronger as the particles get further apart, it has infinite range and there is no place in space, except the origin, where the particles can be free.

Try out the simulation for positive k (attractive potential). Then the system must be bound and the two masses attracted towards each other. This means that even if the wave packets do not have enough energy to reach and bounce off the walls of the box, they must keep returning to the center of mass of the system.

Try out the simulation for negative k (repulsive potential). Then the system cannot be bound and the two masses repel each other more as they get further apart. While this may be physically unrealistic, it should lead to some interesting explosions!

1.5.9 Heavy–Light Particle Collisions

Make one of the particles 10-20 times heavier than the other one and see if your collisions begin to resemble the collision of the light particle with a barrier (as shown on the left in Fig. 1.1).

1.5.10 Two-Particle Density Surface Plots (advanced)

Up until now we have been examining superpositions of the single particle densities for particle one and two, $\rho_1(x)$ and $\rho_2(x)$, superimposed onto the same x axis. These one-particle densities were deduced from the two-particle density $\rho(x_1, x_2)$ by projection, for example,

$$\rho(x_1) = \int_{-\infty}^{+\infty} dx_2 \rho(x_1, x_2) \tag{1.31}$$

The plot shown in Fig. 1.2 is a surface plot of the two particle probability density $\rho(x_1, x_2, t)$ for three values of time t. Looking at these plots should make it clear why we have not been studying them; the extra degree of freedom here makes it hard to see the physics.

Now study $\rho(x_1, x_2, t)$ in 3D for some of the runs that you have already studied in 2D. The purpose is to develop some understanding and intuition about these two-particle functions by relating them to the behaviors found in 2D.

The data used to produce the plots in Fig. 1.2 are from the files `Rho_0.dat, Rho_.5.dat`, and `Rho_1.dat`. These files are produced with `choice = 1` and the full version of the code in *packets.c* (more memory is needed to store these functions). These 3D surface plots were made with *gnuplot*, although there are other tools for making 3D plots of data (even Maple).

The Gnuplot commands for making surface plots are discussed on pages 168-170 of Landau-Fink [14], which we reproduce here.

Gnuplot Surface (3D) Plots

Most of what we have seen in our use of Gnuplot for 2D plots can be extended to 3D or surface plots. The surface plot command is **splot**, and it is used in the same manner as **plot**—with the obvious extension from (x, y) to (x, y, z). As is rather standard for 3D plots, a surface is often specified just by giving its z values for successive rows, for example:

-4.043764 -4.047083 ⋮ -11.000000 -11.000000	Data for 3D plot.

This is typical surface plot data with *strips* of z values separated by spaces. Since there are no explicit x and y values given, Gnuplot labels the x and y axes with the row and column number. Axis labels and titles can be specified explicitly. These data produce *hidden line* surfaces via the commands:

```
gnuplot>  set hidden3d                          Hidden line plot.
gnuplot>  set nokey                             No label.
gnuplot>  splot 'gnu3d.data' w l                Surface plot with lines.
```

While viewing the plot in one of the X Windows, you modify its viewing angle, labels, or scale by entering text in the command window. The command to do this is:

```
gnuplot>  set view rotx, rotz, scale, scalez
```

GNUPLOT VIEWS

rotx	Rotate about x axis 0 to 180^o (default 60).
rotz	Rotate about z axis 0 to 360^o (default 30).
scale	Scale entire plot (default 1.0).
scalez	Scale z axis (default 1.0).

The changes occur when you redraw the plot using the **replot** command. For example, by issuing the commands:

```
gnuplot>  set view 30, 105                      Move over and up.
gnuplot>  replot                                Redraw with the changes.
```

As another example for specifying data, we start with a data file containing the z values for an even 36×16 grid in x and y values. To create our plot we then set the sampling rates. We take **samples** as the number of points on the x axis, and **isosamples** as the corresponding y sampling rate:

```
gnuplot>  set samples 36                                  Number along x axis.
gnuplot>  set isosample 16                                Number along y axis.
gnuplot>  splot 'ip' with lines using z '%f'
```

To now get more realistic, below is a sample session in which (x, y, z) data are read in from the file `ImT.dat`. A 3D surface is then plotted, but without hidden lines removed, along with a contour plot projected onto xy plane.

```
gnuplot>  set nokey
gnuplot>  set view 120, 25
gnuplot>  set xrange [1300:1600]
gnuplot>  set yrange [-100:0]
gnuplot>  set zrange [-30:30]
gnuplot>  set xtics ("1300" 1300, "" 1331.15297, \
    "1400" 1400, "" 1434.57747, "1500" 1500, "1600" 1600)
gnuplot>  set ytics -100, 20, 0
gnuplot>  set ztics -30, 15, 30
gnuplot>  set xlabel "Re E (MeV)"
gnuplot>  set ylabel "Im E (MeV)"
gnuplot>  set param
gnuplot>  set contour
gnuplot>  set cntrparam levels 10
gnuplot>  set terminal postscript eps "Times-Roman" 14
gnuplot>  set label "Im T(E)" at 1450, -100, 60
gnuplot>  set output "ImT.ps"
gnuplot>  splot 'ImT.dat' w l
```

The form of the data is:

```
1300.00000 -0.00100 -0.00001
1303.00000 -0.00100 -0.00000
1306.00000 -0.00100 +0.00000
```

1.5.11 Correlations during Collisions (research project)

(Based on the undergraduate thesis work of David Vediner.) We have solved the Schrödinger equation for the two-particle wave function $\psi(x_1, x_2)$, from which we deduced the two-particle probability density

$$\rho(x_1, x_2, t) = |\psi(x_1, x_2, t)|^2. \tag{1.32}$$

The basic tenets of quantum mechanics (and probability theory) tell us that if particles 1 and 2 were noninteracting or noncorrelated, then the probability of finding particle 1 at x_1 and particle 2 at x_2 is just the products of the individual probabilities,

$$\rho(x_1, x_2, t)|\text{uncorr} = \rho_1(x_1, t)\rho_2(x_2, t). \tag{1.33}$$

Since the two particles in our simulation are interacting, we do not expect (1.33) to be true, especially at the times during which the particles interact. Accordingly, we define the correlation function as the difference between the full two-particle density and the product

of the single-particle densities (which would be the two-particle density if the particles were uncorrelated):

$$C(x_1, x_2, t) = \rho(x_1, x_2, t) - \rho_1(x_1, t)\rho_2(x_2, t). \tag{1.34}$$

For just one set of parameters, produce output files of $\rho(x_1, x_2, t)$, $\rho_1(x_1, t)\rho_2(x_2, t)$, and $C(x_1, x_2, t$. For for times before, during, and after a collision, compared the 3D surface plots of each. Comment on whether $C(x_1, x_2, t$ shows the particle-particle correlations. If you like, you can modify our previous scripts and procedures to make movies of these functions. Sample are on the Web site [1].

1.5.12 2D Collisions (research project)

Generalize method. The potential need not be symmetric or central.

1.6 Code Listing Square.c

```
/*
*************************************************************************
*  squareSym.c: solution of time-dependent Schroedinger Equation       *
*             for wave packet - wave packet scattering                   *
*             This is lite version which uses less  memory             *
*             Square potential *and* symmetrization included           *
*  Copyright 1999, R. Landau, and Jon Maestri, Oregon State University *
*                                                                       *
*  supported by: US National Science Foundation, Northwest Alliance    *
*                for Computational Science and Engineering (NACSE),    *
*                US Department of Energy                               *
*************************************************************************
*/


#include <stdio.h> #include <math.h> #include <time.h>

#define xDim 201               /* number space pts, must be odd
for simpson */

FILE *out1, *out2, *out3a, *out3b, *out3c, *out7a, *out7b, *out7c;
FILE *out8, *out10, *out11, *out12, *out13, *out14, *input;

/*               Prototypes
*/ /* void Potential(); */ void Initialize(void); void
SolveSE(void); void Output(); void Probability();

 /*              Global Variables                                        */
double RePsi[xDim][xDim][2], ImPsi[xDim][xDim][2]; double
rho_1[xDim], rho_2[xDim], SumRho[xDim]; double w[3]; double a1,
a2, a3, a4, alpha, dx, dt, Er, Ei, Eri, Eii; double k1, k2, m1,
m2, dm1, dm2, dm12, dm22,dxx2, dtx, con, con2; double Ptot,
Ptot_i; double sol, sig,   Rho, v, vmax, x01, x02, y; int choice,
symmetry,  Nt, N1, N2, nprint; time_t timei, timef;

/*************************************************************************
*                      MAIN                                              *
*************************************************************************/
 main()
{

int i, j; time_t timei, timef;

timei = time(NULL);        /* start  program timing*/

/* inititialize variables  */

input = fopen("in.dat", "r");
fscanf(input, "%d %lf %lf %lf %lf", &Nt, &k1, &k2, &dx, &dt);
fscanf(input, "%lf %lf %lf  %lf  %lf", &sig, &x01, &x02, &m1, &m2);
fscanf(input, "%lf %lf %d %d", &vmax, &alpha, &nprint, &symmetry);
fclose(input);

/* Some constant combos needed in prog */

N1 = xDim-2; /* Max number in sum */ N2 = xDim-2; dm1 = 1./m1; dm2
= 1./m2; dm12 = 0.5/m1; dm22 = 0.5/m2; dxx2 = dx*dx; dtx =
```

```
dt/dxx2; con = -1./(2.*dxx2); con2 = (dm1+dm2)*dtx;

/* Open files */

out2 = fopen("V.dat", "wb"); out8 = fopen("SumRho.dat", "wb");
out10 = fopen("logP_t.dat", "wb"); out11 = fopen("E_t.dat", "wb");
out12 = fopen("logE_t.dat", "wb"); out14 = fopen("params.dat",
"wb");

/* Choice Menu:                                *
 * choice = 0, view Rho   (3d)                 *
 * choice = 1, view rho_1 (2d)                 *
 * choice = 2, view corr(x) (2d)  (experimental) *
 * choice = 3, view rho_1 + rho_2              */

choice = 1;

/* Symmetry Menu:
 * symmetry = 0, do not impose syymerty        *
 *          = +1, symmetrize wave function       *
 *          = -1, antisymmetrize wave function    */

/* Print  initial values to the params file */

fprintf(out14, "choice =  %d\n", choice);
fprintf(out14, "symmetry =  %d\n", symmetry);
fprintf(out14, "k1 =  %lf\n", k1);
fprintf(out14, "k2 =  %lf\n", k2);
fprintf(out14, "dx =  %lf\n", dx);
fprintf(out14, "dt =  %e\n", dt);
fprintf(out14, "sig =  %lf\n", sig);
fprintf(out14, "alpha =  %lf\n", alpha);
fprintf(out14, "x01 =  %lf\n", x01);
fprintf(out14, "x02 =  %lf\n", x02);
fprintf(out14, "N1 =  %d\n", N1);
fprintf(out14, "N2 =  %d\n", N2);
fprintf(out14, "m1 =  %lf\n", m1);
fprintf(out14, "m2 =  %lf\n", m2);
fprintf(out14, "vmax =  %lf\n", vmax);
fprintf(out14, "nprint =  %d\n", nprint);
fprintf(out14, "xDim =  %d\n", xDim);
fprintf(out14, "Nt =  %d\n", Nt);

/* Initialize Simpson integration weights */ /*            [end
values not used if psi(ends) = 0] */

w[0] = dx/3.; w[1] = 4.*dx/3.; w[2] = 2.*dx/3.;

/* initialize Potential thru all space Potential();*/


/* initialize  Wave packet */ Initialize();


/* Solve Schrodinger equation */ SolveSE();

/*end the time of program */

timef = time(NULL);
fprintf(out14, "\nElapsed time is: %ld seconds\n", timef-timei);

fclose(out8); fclose(out10); fclose(out11); fclose(out12);
fclose(out14);

}

/*************************************************************************
 *      Energy                                                           *
 *************************************************************************/

void Energy(int n){

double ai1, ai2, tmp, Erel; int i,j,k,p;

/* calculate total energy of system */

Er = 0.; Ei = 0.; p = 1;        /* wf zero at boundaries,
therefore no p,k = 0 simpson terms */ for (i = 1; i <= N1; i++){
    k = 1;
    if (p == 3)p = 1;
    for (j = 1; j <= N2; j++){
```

```
        if (k == 3)k = 1;
        /* square well in line */
        if(alpha >= abs(i-j)*dx){v=vmax;}else{v=0.;}
        a1 = -2*(dm1+dm2+dx*dx*v)*(RePsi[i][j][1]*RePsi[i][j][1]
            +ImPsi[i][j][1]*ImPsi[i][j][1]);
        a2 = dm1*(RePsi[i][j][1]*(RePsi[i+1][j][1]+RePsi[i-1][j][1])
            +ImPsi[i][j][1]*(ImPsi[i+1][j][1]+ImPsi[i-1][j][1]));
        a3 = dm2*(RePsi[i][j][1]*(RePsi[i][j+1][1]+RePsi[i][j-1][1])
            +ImPsi[i][j][1]*(ImPsi[i][j+1][1]+ImPsi[i][j-1][1]));
        ai1 = dm1*(RePsi[i][j][1]*(ImPsi[i+1][j][1]+ImPsi[i-1][j][1])
             -ImPsi[i][j][1]*(RePsi[i+1][j][1]+RePsi[i-1][j][1]));
        ai2 = dm2*(RePsi[i][j][1]*(ImPsi[i][j+1][1]+ImPsi[i][j-1][1])
             -ImPsi[i][j][1]*(RePsi[i][j+1][1]+RePsi[i][j-1][1]));
        Er = Er + w[k]*w[p]*con*(a1 + a2 + a3);
        Ei = Ei + w[k]*w[p]*con*(ai1 + ai2);
        k = k+1;
        }
    p = p+1;
    }
/* Normalize */
       Er = Er/Ptot_i;  Ei = Ei/Ptot_i;
/* make sure number is not too small, gnuplot can't take them */
    if (fabs(Er) <= 1.e-20)Er = 0.;

/* print to "EvsT.dat" file */
    fprintf(out11, "%d %e\n", n, Er);

/* find relative Energy */
    tmp  = fabs((Er/(Eri))-1.);
    if (tmp  != 0.)Erel = log10(tmp);

/* print to "ErelvsT.dat"*/
   fprintf(out12, "%d %e\n", n, Erel);

return; }

/*************************************************************************
 *           Initialize                                                  *
 *************************************************************************/

void Initialize() {

double x1, x2, ww, X01, X02; double ai1, ai2; int i, j, k, p;

/* determine omegas in terms of wave vectors */

ww = (k1*k1/(2.*m1) + k2*k2/(2.*m2))*0.5*dt;

/* Initialize  wave function, scale initial positions to box
size*/

x1 = 0.; X01 = x01*xDim*dx; X02 = x02*xDim*dx; for (i = 1; i <=
N1; i++){
    x1 = x1+dx;
    x2 = 0.;
    for (j = 1; j <= N2; j++){
        x2 = x2+dx;
        y = k1*x1-k2*x2;
        y = y-ww;
        a1 = (x1-X01)/sig;
        a2 = (x2-X02)/sig;
        a4 = exp(-(a1*a1+a2*a2)/2.);
        RePsi[i][j][0] = a4* cos(y);
        ImPsi[i][j][0] = a4* sin(y);
        }

    }

/* Set wf to zero on boundary (should never be called anyway) */

/*  at x1 edges RePsi is zero */
    for (j = 0; j <= N1+1; j++){
        RePsi[N1+1][j][0] = 0.;
        RePsi[0][j][0] = 0.;
        }
/*  at x2 edges RePsi is zero */
    for (i = 1; i <= N2; i++){
        RePsi[i][N2+1][0] = 0.;
        RePsi[i][0][0] = 0.;
        }
```

```
/* Find the initial (unnormalized) energy at roughly t = 0 */

Eri = 0.; Eii = 0.; p = 1; for (i = 1; i <= N1; i++){
    k = 1;
    if (p == 3)p = 1;
    for (j = 1; j <= N2; j++){
        if (k == 3)k = 1;
        /* special square well */
        if(alpha >= abs(i-j)*dx){v=vmax;}else{v=0.;}
        a1 = -2*(dm1+dm2+dx*dx*v)*(RePsi[i][j][0]*RePsi[i][j][0]
             +ImPsi[i][j][0]*ImPsi[i][j][0]);
        a2 = dm1*(RePsi[i][j][0]*(RePsi[i+1][j][0]+RePsi[i-1][j][0])
             +ImPsi[i][j][0]*(ImPsi[i+1][j][0]+ImPsi[i-1][j][0]));
        a3 = dm2*(RePsi[i][j][0]*(RePsi[i][j+1][0]+RePsi[i][j-1][0])
            +ImPsi[i][j][0]*(ImPsi[i][j+1][0]+ImPsi[i][j-1][0]));
        ai1 = dm1*(RePsi[i][j][0]*(ImPsi[i+1][j][0]+ImPsi[i-1][j][0])
            -ImPsi[i][j][0]*(RePsi[i+1][j][0]+RePsi[i-1][j][0]));
        ai2 = dm2*(RePsi[i][j][0]*(ImPsi[i][j+1][0]+ImPsi[i][j-1][0])
            -ImPsi[i][j][0]*(RePsi[i][j+1][0]+RePsi[i][j-1][0]));
        Eri = Eri + w[k]*w[p]*con*(a1 + a2 + a3);
        Eii = Eii + w[k]*w[p]*con*(ai1 + ai2);
        k = k+1;
        }
    p = p+1;
    }

/* print initial Energy to "params" file */
    fprintf(out14, "E (unnormalized) initial = %lf\n", Eri);
    fprintf(out14, "E imaginary intitial = %lf\n", Eii);
}


/*************************************************************************
 *      Output                                                           *
 *************************************************************************/

void Output(n){

int i, j; char s[] = "run.00001";


/* Create a unique file name with data for each time step */

    if (n < 10){
        s[8] =  n+48;
        }
    if (n < 100 && n > 9){
        s[7] = (n/10)+48;
        s[8] =  (n%10)+48;
        }
    if (n < 1000 && n > 99){
        s[6] = (n/100)+48;
        s[7] =  (((n%100))/10)+48;
        s[8] =  (n%10)+48;
        }
    if (n < 10000 && n > 999){
        s[5] = (n/1000)+48;
        s[6] = (((n%1000))/100)+48;
        s[7] =  (((n%100))/10)+48;
        s[8] =  (n%10)+48;
        }
    if (n < 100000 && n > 9999){
        s[4] = (n/10000)+48;
        s[5] = (((n%10000))/1000)+48;
        s[6] = (((n%1000))/100)+48;
        s[7] =  (((n%100))/10)+48;
        s[8] =  (n%10)+48;
        }

/* Print probability vs position data at each time */

if (choice == 0){ if (n == 1){ out1 = fopen(s,"wb");
    for (i = 1; i <= N2; i++){
        for (j = 1; j <= N2; j++){
            Rho = RePsi[i][j][0]*RePsi[i][j][1]
                 +ImPsi[i][j][0]*ImPsi[i][j][0];
/* Impose symmetry or antisymmetry */
           Rho = Rho + symmetry*(RePsi[i][j][0]*RePsi[j][i][1]
                     + ImPsi[i][j][0]*ImPsi[j][i][0]);
            if (Rho < 1.e-20)Rho = 1.e-20;
            fprintf(out1, "%e\n", Rho);
```

```
            }
        fprintf(out1, "\n");
        }
    fclose(out1);
    }
out1 = fopen(s,"wb");
    for (i = 1; i <= N2; i++){
        for (j = 1; j <= N2; j++){
            Rho = RePsi[i][j][0]*RePsi[i][j][1]
                 +ImPsi[i][j][0]*ImPsi[i][j][0];
/* Impose symmetry or antisymmetry */
           Rho = Rho + symmetry*(RePsi[i][j][0]*RePsi[j][i][1]
                     + ImPsi[i][j][0]*ImPsi[j][i][0]);
            if (Rho < 1.e-20)Rho = 1.e-20;
            fprintf(out1, "%e\n", Rho);
            }
        fprintf(out1, "\n");
        }
    fclose(out1);
    }
if (choice == 1){ out1 = fopen(s,"wb");
    for (i = 1; i <= N1; i++){
        if (rho_1[i] < 1.e-20)rho_1[i] = 1.e-20;
        if (rho_2[i] < 1.e-20)rho_2[i] = 1.e-20;
        fprintf(out1, "%d %e %e\n", i, rho_1[i], rho_2[i]);
        }
    fclose(out1);
    }
/* if (choice == 2){ out1 = fopen(s,"wb");
    for (i = 1; i <= N2; i++){
        if (rho_1[i] < 1.e-20)rho_1[i] = 1.e-20;
        if (rho_2[i] < 1.e-20)rho_2[i] = 1.e-20;
        fprintf(out1, "%d %e %e\n", i, corr[i]);
        }
    fclose(out1);
    }
*/ if (choice == 3){ out1 = fopen(s,"wb");
    for (i = 1; i <= N1; i++){
        sol = rho_2[i]+rho_1[i];
        if (sol < 1.e-20)sol = 1.e-20;
        fprintf(out1, "%d %e\n", i, sol);
        }
    fclose(out1);
    }

/* Print out some data sets for time = 1 */

if (n == 1){
    /* print out potential */
        for (i = 1; i <= N1; i = i+10){
        for (j = 1; j <= N2; j = j+10){
          /* special square well */
           if (alpha>=abs(i-j)*dx){v=vmax;}else{v=0.;}
            fprintf(out2, "%e\n", v);
            }
        fprintf(out2, "\n");
        }

    }
    fclose(out2);
}

/*************************************************************************
 *      Potential- now square                                            *
 *************************************************************************

void Potential() {

double tmp; int i, j;

tmp = alpha/dx;

for (i = 0; i <= N1+1; i++){
    for (j = 0; j <= N2+1; j++) {
         if (abs(i-j)*dx <= alpha) v[i][j] = vmax;
         else v[i][j] = 0.;
                } }

}

*/
```

```
/*************************************************************************
 *      Probability                                                      *
 *************************************************************************/

void Probability(int n){ double tmp, Prel, x; int i,j,k,p;


/* initialize the single probability arrays to zero */ for (i = 0;
i <= N1+1; i++){
    rho_1[i] = 0.;
    rho_2[i] = 0.;
/*      corr[i]  = 0.;   */ }

/* Normalize Rho, important for Correlation functions */ Ptot =
0.; Prel = 0.; p = 1; for (i = 1; i <= N1; i++){
    k = 1;
    if (p == 3)p = 1;
    for (j = 1; j <= N2; j++){
          Rho = RePsi[i][j][0]*RePsi[i][j][1]
                +ImPsi[i][j][0]*ImPsi[i][j][0];
/* Impose symmetry or antisymmetry */
           Rho = Rho + symmetry*(RePsi[i][j][0]*RePsi[j][i][1]
                     + ImPsi[i][j][0]*ImPsi[j][i][0]);
        if (k == 3)k = 1;
        (Ptot) = (Ptot) + w[k]*w[p]*Rho;
        k = k+1;
             }
    p = p+1;
          }

if (n == 1)(Ptot_i) = (Ptot);          /* Assign  initial
probability */ /* Renormalize Rho, before inlined for (i = 1; i <=
N1; i++){for (j = 1; j <= N2; j++){ Rho[i][j] =
Rho[i][j]/(Ptot);}}*/

/* Integrate out 1D probabilites from 2D */

p = 1; for (i = 1; i <= N1; i++){
    k = 1;
    if (p == 3)p = 1;
    for (j = 1; j <= N2; j++){
        if (k == 3)k = 1;
        Rho = RePsi[i][j][0]*RePsi[i][j][1]
                +ImPsi[i][j][0]*ImPsi[i][j][0];
/* Impose symmetry or antisymmetry */
           Rho = Rho + symmetry*(RePsi[i][j][0]*RePsi[j][i][1]
                     + ImPsi[i][j][0]*ImPsi[j][i][0]);
        rho_1[i] = rho_1[i] + w[k]*Rho;
        Rho = RePsi[j][i][0]*RePsi[j][i][1]
                +ImPsi[j][i][0]*ImPsi[j][i][0];
/* Impose symmetry or antisymmetry */
           Rho = Rho + symmetry*(RePsi[j][i][0]*RePsi[i][j][1]
                     + ImPsi[j][i][0]*ImPsi[i][j][0]);
        rho_2[i] = rho_2[i] + w[k]*Rho;
            k = k+1;
        }
/* sum 1D probabilities and print to file */
    SumRho[i] = rho_1[i] + rho_2[i];
    if (fabs(SumRho[i]) < 1.e-20) SumRho[i] = 0.;
    if (i%10 == 0) fprintf(out8, "%e\n", SumRho[i]);
    p = p+1;
    }
fprintf(out8, "\n");

/* find relative probability and print it to file */

    tmp  = fabs(((Ptot)/(Ptot_i ))-1.);
    if (tmp  != 0.) Prel= log10(tmp);
    fprintf(out10, "%d %e\n", n, Prel);

/* Determine 1 particle correlation function *
           for i+j fixed (on other diagnol) vs x = i-j

if (n == Nt/2){ for (i = 1; i <= N1; i = i+5)
   {
    j =  N1+1-i;
    x = i-j;
    x = fabs(x);
      if ((rho_1[i] != 0.)&&(rho_2[j] != 0.))
      corr[i] =  Rho[i][j]/rho_1[i]/rho_2[j];
    if ((rho_1[j] != 0.)&&(rho_2[i] != 0.))
```

```
      corr[i] =  corr[i]+Rho[j][i]/rho_1[j]/rho_2[i];
    if (corr[i] != 0.) corr[i] = log10(fabs(corr[i]));
    fprintf(out13, "%lf %e\n", x, corr[i]);
    }
fprintf(out13, "\n"); } */ }

/*************************************************************************
 *      SolveSE                                                          *
 *************************************************************************/

void SolveSE(){

/* solve time dependent Schroedinger equation */

int n, i, j; char s[] = "ru21.0001"; FILE *out;

/* Start of the loop that finds the probability at each time step
*/

for (n = 1; n <= Nt;n++){
    /*  compute real part of  WF  */
    for (i = 1; i <= N1; i++){
        for (j = 1; j <= N2; j++){
          /* special square well */
              if (alpha>=abs(i-j)*dx){v=vmax;}else{v=0.;}
            a2 = dt*v*ImPsi[i][j][0]+con2*ImPsi[i][j][0];
            a1 = dm12*(ImPsi[i+1][j][0]+ImPsi[i-1][j][0])
               +dm22*(ImPsi[i][j+1][0]+ImPsi[i][j-1][0]);
            RePsi[i][j][1] = RePsi[i][j][0]-dtx*a1+2.*a2;


        /* compute probability density (old, now Rho inlined)
            if (n%(nprint) == 0|n == 1|n == Nt/2|n == Nt)
            Rho[i][j] = RePsi[i][j][0]*RePsi[i][j][1]
                 +ImPsi[i][j][0]*ImPsi[i][j][0];       */
            }
        }


/*Integrate out all the different Probabilities */
    if (n%(nprint) == 0|n == 1|n == Nt/2|n == Nt) Probability(n);


    /*       imaginary part of wave packet is next    */
    for (i = 1; i <= N1; i++){
        for (j = 1; j <= N2; j++){
         /*special square well  */
           if (alpha>=abs(i-j)*dx){v=vmax;}else{v=0.;}
            a2 = dt*v*RePsi[i][j][1]+con2*RePsi[i][j][1];
            a1 = dm12*(RePsi[i+1][j][1]+RePsi[i-1][j][1])
                +dm22*(RePsi[i][j+1][1]+RePsi[i][j-1][1]);
            ImPsi[i][j][1] = ImPsi[i][j][0]+dtx*a1-2.*a2 ;

            }
        }

/* Find  Energy  */
    if (n%(nprint) == 0|n == 1) Energy(n);

/* Generate data files (most of them) */
    if (n%(nprint) == 0|n ==  1|n == Nt/2|n == Nt) Output(n);
        /* new iterations are now the old ones, recycle */
        for (i = 1; i <= N1; i++){
            for (j = 1; j <= N2; j++){
                ImPsi[i][j][0] = ImPsi[i][j][1];
                RePsi[i][j][0] = RePsi[i][j][1];
                }
        }
    }
}
```

Bibliography

[1] *Movies of wave packet–wave packet Quantium Scattering*, `href://nacphy.physics.orst.edu/ComPhys/PACKETS/`

[2] J.J.V. Maestri, R.H. Landau, and M.J. Páez, *Two-particle Schrödinger equation animations of wave packet-wave packet scattering*, Am. J. Phys. **68**, 1113-1119 (2000).

[3] L.I. Schiff, *Quantum Mechanics* (third edition), McGraw-Hill, New York (1968), p 106.

[4] A. Goldberg, H. M. Schey, and J. L. Schwartz, *Computer-Generated Motion Pictures of One-Dimensional Quantum-Mechanical Transmission and Reflection Phenomena,* Amer. J. Phys., **35**, 177–186 (1967).

[5] A. Askar and A.S. Cakmak, *Explicit Integration Method for the Time-Dependent Schrödinger Equation for Collision Problems,* J. Chem. Phys., **68**, 2794–2798 (1978).

[6] P.B. Visscher, *A Fast Explicit Algorithm for the Time-Dependent Schrödinger Equation,* Computers In Physics, 596–598 (Nov/Dec 1991).

[7] S. Brandt and H.D. Dahmen, *Quantum Mechanics on the Personal Computer,* Chapt. 5, Springer-Verlag, Berlin, 82–92 (1990).

[8] R.H. Landau and M.J. Paez, *Computational Physics, Problem Solving With Computers,* Wiley, New York, 399–408 (1997).

[9] S.E. Koonin, *Computational Physics,* Benjamin, Menlo Park, 176–178 (1986).

[10] N.J. Giordano, *Computational Physics,* Prentice Hall, Upper Saddle River, 280–299 (1997).

[11] Our animations are animated *gifs* that can be viewed with any Web browser, or viewed and controlled with a movie player such as *Quick Time*. To create them, we have the C code *packets.c* output files of wave function data for each time. We plot each data file with the Unix program *gnuplot* to produce one frame, and then convert the plots to gif files. We then use *gifmerge* to merge the frames into an animation. Further information and instructions for making movies using different operating systems and formats can be found on the Landau Research Group Web pages[1, 12].

[12] *Visualization Of Scientific Data*, `http://nacphy.physics.orst.edu/DATAVIS/datavis.html`

[13] GIFMerge Rev 1.30 (C) 1991,1992 by Mark Podlipec, improvements by Rene K. Mueller 1996; see too http://www.iis.ee.ethz.ch/~kiwi/gifmerge.html.

[14] R.H. Landau and P.J. Fink, *A Scientist's and Engineer's Guide to Workstations and Supercomputers*, Wiley, N.Y. 1993.