Bose-Einstein Condensates in 'Magic' Optical Traps
PHS3350 Physics Project Report

Chris Billington[*]

Monash University

Supervisors

Prof. Kristian Helmerson

Dr. Lincoln Turner

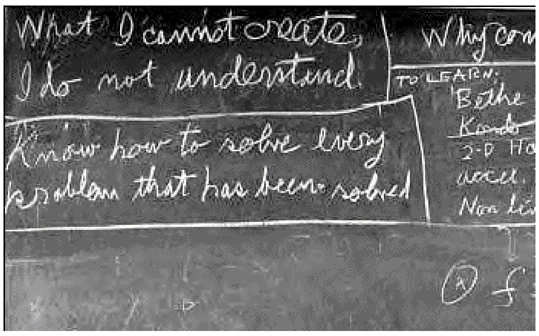
(Dated: October 23, 2009)

Abstract

It is possible to use the optical dipole force to trap atoms at the focus of a laser, and Bose-Einstein condensates (BECs) can be held in optical traps this way. For some wavelengths of light, the dipole force is zero and atoms cannot be trapped. Called 'magic wavelengths', they depend on the particular type of atom in question, and so one can be chosen which traps one species of atom, but not another. In this project, I investigated the 1D scattering of a 'bullet' BEC which sees no optical trap off a 'target' BEC of a different species, trapped at the focus of a laser. The two BECs obey the Gross-Pitaevskii equation and may attract or repel each other depending on the sign of the interspecies scattering length. I considered repulsive interactions only, and so the bullet BEC sees the target as a potential barrier, one with which it can interact. The main result of these simulations was a determination of the bullet's reflection coefficient as a function of its incident kinetic energy. Rather than the smooth, monotonically decreasing curve that one would obtain for scattering off a static potential, the interaction with the barrier gave rise to resonances of higher than normal reflection coefficient. This was attributed to the target BEC being excited into discrete 'breathing modes' in its trap. I used the XMDS simulation software package to numerically solve the differential equations, and made what I believe to be a significant improvement to one of the algorithms commonly used for simulations of BECs.

[*]Electronic address: `cjbil1@student.monash.edu`

I. OVERVIEW



What I cannot create, I do not understand.

Know how to solve every problem that has been solved.

-Written on Richard Feynman's blackboard at the time of his death.

This project began with background reading on several topics, including the interaction of atoms with radiation, and scattering theory. My approach for scattering theory was (as perhaps hinted at by the above quote) to understand concepts by being able to apply them numerically and calculate important quantities. To this end I wrote programs to calculate radial wavefunctions, scattering phase shifts, scattering lengths and bound states for arbitrary potentials. Although these programs aren't required for the BEC simulations, they are included here in the hope that they may be useful.

Consideration of the interaction of atoms with radiation allows derivations of both the optical scattering force[1] and the optical dipole force. The latter can be used to trap atoms at the focus of a laser, and its wavelength dependence allows for the existence of magic wavelengths at which there is no net force on the atoms.

My reading on scattering theory was based on the method of partial waves. In the low energy limit this allows the calculation of interatomic scattering lengths, on which the nonlinear term in the Gross-Pitaevskii equation (GPE) depends. When the interatomic potential has or is close to having a bound state near zero energy, the scattering length can be extremely large, a phenomenon known as 'zero energy resonance'. When a bound state exists near zero energy but in a *different spin state* of the atom pair, the scattering length can

[1] Essential for the laser cooling used in BEC production.

also be very large, provided the two spin states are coupled through the hyperfine interaction. This is known as a Feshbach resonance [1] and allows tuning of the scattering length, since one can shift the relative energies of the two spin states with an applied magnetic field.

The main focus of the project was the simulation of Bose-Einstein condensates. After failed attempts at using Python to run these simulations, I was introduced to the industry standard software package XMDS (eXtensible Multi-Dimensional Simulator), as well as standard techniques for calculating the groundstate and time evolution of the Gross-Pitaevskii equation (GPE). Many 'measurements' were taken of the evolution of the coupled system of BECs in order to determine the properties of their interaction.

A Python script was written to run the scattering simulation over many incident energies, and produce videos of them. The reflection coefficient for each simulation could be read off these videos, to determine the reflection coefficient's dependence on the collision energy. The results showed that at certain energies there were local peaks in the reflection coefficient, and this is attributed to an energy exchange due to the target BEC being excited into discrete modes of its potential well. The resulting loss of kinetic energy of the bullet BEC would cause more of it to be reflected than when off resonance.

II. DIPOLE FORCE AND MAGIC WAVELENGTHS

My reading on the dipole force started with the purely classical model in which one has the electron as a charged sphere orbiting the nucleus and interacting with classical electromagnetic radiation (Foot, chapter 9.6 [2]). This enabled a determination of an expression for the dipole force. Quantum mechanically, the dipole force and scattering force can be derived using a two level atom in a classical light field. The expressions thus arrived at are:

$$F_{scatt} = \hbar k \frac{\Gamma}{2} \frac{\Omega^2/2}{\delta^2 + \omega^2/2 + \Gamma^2/4}, \tag{1}$$

and

$$F_{dipole} = -\frac{\hbar\delta}{2} \frac{\Omega}{\delta^2 + \omega^2/2 + \Gamma^2/4} \frac{\partial\Omega}{\partial z}, \tag{2}$$

where δ is the frequency detuning from resonance, Γ is the linewidth, or equivalantly the reciprocal of the decay time of the upper state, ω, k and z are the angular frequecy, wavenumber, and propagation direction of the incident light, and Ω is the Rabi frequency:

$$\Omega = \frac{e\mathbf{E_0}}{\hbar}\langle 1|x|2\rangle, \tag{3}$$

where x is the polarization direction of the electric field. To understand these derivations I read through chapter 7 of Foot [2], where the interaction of a two level atom with radiation is studied. The derivation of the oscillations of the populations of the two states, called Rabi flopping, made use of the rotating wave approximation [2] with the main result that at resonance the upper state population varies as:

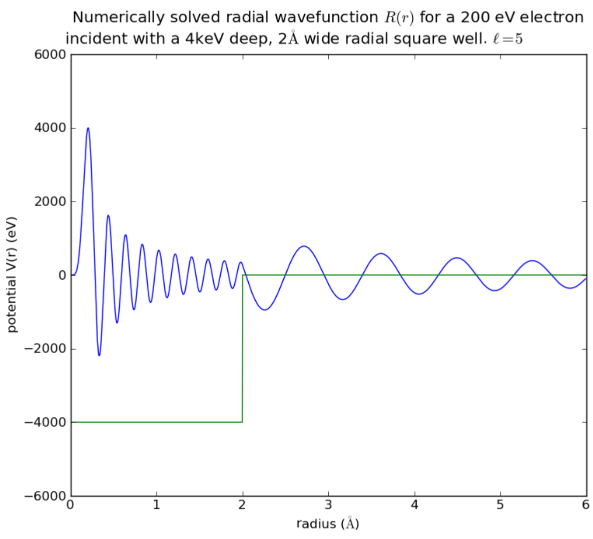
$$|c_2(t)|^2 = \sin\left(\frac{\Omega t}{2}\right). \tag{4}$$

The quantity $\langle 1|x|2\rangle$ in eqn (3) is called the dipole moment, and depends on the shapes of the wavefunctions of the two states $|1\rangle$ and $|2\rangle$. Of course, real atoms have more than one transition, and so there are many dipole moments. As can be seen in eqn (2), the sign of the dipole force depends on the sign of the detuning from resonance δ, and so a laser that is below resonance for one transition and above resonance for another exerts two opposing dipole forces. If they are equal and opposite forces, then there is *no net dipole force* on the atom (provided there are no further transitions involved). For example, the fine structure of the Sodium D line gives rise to two transition dipole moments that are almost identical. If one shines a laser with a wavelength halfway between those of the two lines, it will be equally as off resonance for both, but with a different sign. Since the dipole moments are the same, this gives rise to zero net force. This is a so called 'magic wavelength' of Sodium.

III. SCATTERING THEORY

I drew on parts of chapter 13 of Bransden and Joachain 1989 [3] and chapter 12 of Bransden and Joachain 2005 [4] to gain an understanding of the time independent scattering of particles off static potentials. This required the concepts of partial wave expansions, scattering lengths and phase shifts.

[2] which ignores high frequency oscillations of the populations of the two levels, instead looking at the populations on a longer timescale



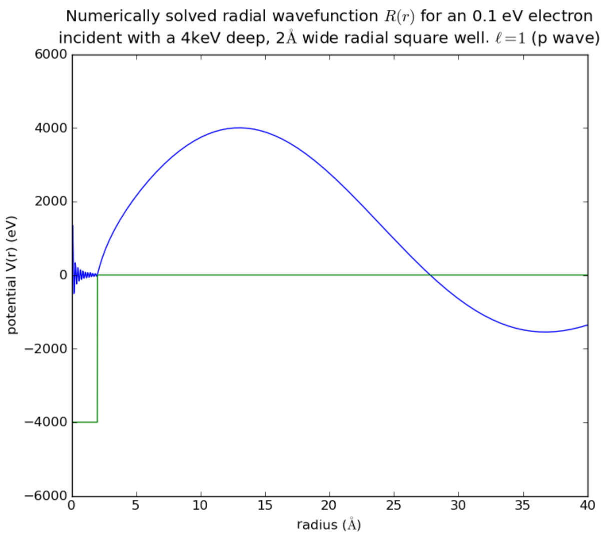


FIG. 1: (Left) A radial wavefunction $R(r) = r^{-1}u(r)$ with $u(r)$ numerically determined by the Python program. Features evident are the exclusion from $r = 0$ due to the nonzero angular momentum, and the increase in wavelength in the region outside the well due to the electron having less kinetic energy there.

FIG. 2: (Right) Another radial wavefunction, with a much smaller wavenumber for the incident electron. Since the wavefunction has a much greater amplitude outside the well, the external solution must be practically zero (compared to its amplitude) just outside the well. This already hints at the concept of a scattering length for small wavenumbers, since the wavefunction has to be close to zero there regardless of what happens in the interior. The small amplitude within the well is due to the centrifugal barrier which exists for $\ell \neq 0$.

I wrote programs in Python to numerically solve the equations of scattering and output the phase shifts that each partial wave undergoes in scattering off an arbitrary potential. Since the square well potential was one which was given an analytic treatment in both books, I had the program numerically determine phase shifts for square well scattering, and compared these to the analytic solutions. The methods by which the program determines phase shifts is outlined below.

A. Numerically solving the radial Schrodinger equation

The first step in determining the phase shifts was solving the time independent radial Schrodinger equation for the ℓth partial wave:

$$\left[\frac{d^2}{dr^2} - \frac{\ell(\ell+1)}{r^2} - U(r) + k^2\right] u_\ell(k,r) = 0, \tag{5}$$

where $U(r)$ is the reduced potential $U(r) = \frac{2mV(r)}{\hbar^2}$, k is the wavenumber of the incident particle, and $u_\ell(k,r) = rR_\ell(k,r)$ is the radial wavefunction scaled by r. This equation has the asymptotic solution for small r (independent of $U(r)$):

$$u_\ell(k,r) \propto r^{\ell+1}. \tag{6}$$

The differential equation can be written as two coupled first order DEs [5] for two functions: $u_\ell(k,r)$, and its derivative, which we'll call $v_\ell(k,r)$:

$$\frac{d}{dr}v_\ell(k,r) = \left[\frac{\ell(\ell+1)}{r^2} + U(r) - k^2\right] u_\ell(k,r), \tag{7}$$

and

$$\frac{d}{dr}u_\ell(k,r) = v_\ell(k,r). \tag{8}$$

Given initial values for u and v, these two equations can be numerically solved simultaneously using, say, the 4th order Runge-Kutta method (Numerical Recipes, chapter 17 [5]).

A quick look at the equations however reveals that we'll run into serious problems if we attempt to integrate from $r = 0$. Therefore the numerical integration program instead starts at a "small" radius (taken to be a single step size in the numerical integration) where the asymptotic solution to $u_\ell(k,r)$ is assumed to hold. Differentiating the asymptotic solution and evaluating it at the same point gives an initial condition for $v_\ell(k,r)$ also.

One might worry that the asymptotic solution for u, (and hence for v also) is only defined up to an arbitrary multiplicative constant. This is ok, since the entire wavefunction is only defined up to a multiplicative constant, and we don't need it to be normalized to proceed later.

The initial conditions the program uses are therefore:

$$u_\ell(k,\Delta r) = (\Delta r)^{\ell+1}, \tag{9}$$

and

$$v_\ell(k,\Delta r) = (\ell+1)(\Delta r)^{\ell}, \tag{10}$$

where Δr is the integration step size. This is the smallest value of r the program considers, and when it outputs an array for its solutions, these are the first values of u and v.

From these initial values, the solver function then steps along at a user defined step size until it reaches the input maximum r value, then outputs arrays of u, v, and r. Examples of radial wavefunctions determined this way can be seen in Fig 1 and Fig 2. The output is then passed to the "boundary matching" program, which determines the actual phase shifts.

B. Boundary matching and phase shifts

The numerical integration should end at some point a outside the radius of the potential's influence. There, the values of u and v can be used to determine u's logarithmic derivative (note this gets rid of any arbitrary constant out the front of u and v!):

$$\beta_\ell(k) = \left[\frac{1}{u}\frac{du}{dr}\right]_{r=a} = \frac{v_a}{u_a}. \tag{11}$$

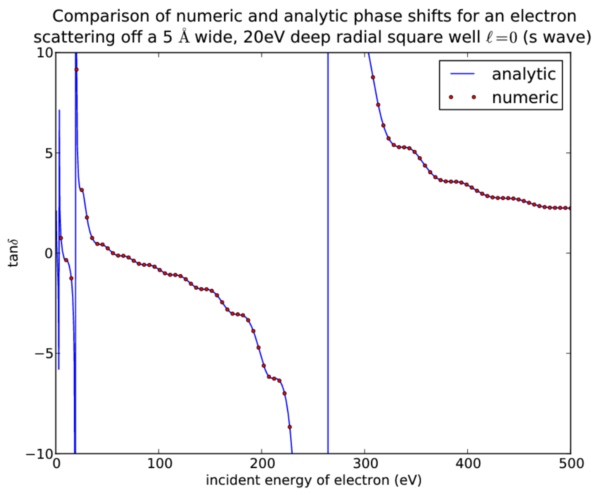
The logarithmic derivative of the exterior solution for $u(r \geq a)$ is known analytically, and can be rearranged to make its only unknown parameter ($\tan\delta$, where δ is the phase shift) the subject. This gives an expression for $\tan\delta$ in terms of the logarithmic derivative of the exterior solution at a. Then by requiring that the logarithmic derivatives of the numerically determined solution and the exterior solution are *the same* at a, we can write that expression in terms of the numberically determined logarithmic derivative $\beta_\ell(k)$ at a:

$$\tan\delta_\ell(k) = \frac{ka\, j'_\ell(ka) + [1 - a\beta_\ell(k)]j_\ell(ka)}{ka\, n'_\ell(ka) + [1 - a\beta_\ell(k)]n_\ell(ka)}, \tag{12}$$

where j_ℓ and n_ℓ are the spherical Bessel and Neumann functions. This is equation 12.98 in Bransden and Joachain 2005, and allows the determination of $\tan\delta_\ell(k)$ whenever we have the wavefunction and its derivative at an exterior radius.

An equivalent way of doing things is with the wavefunction $R(r) = r^{-1}u(r)$ and its derivative, instead of u. In that case R's logarithmic derivative is:

$$\gamma_\ell(k) = \left[\frac{1}{R}\frac{dR}{dr}\right]_{r=a}, \tag{13}$$



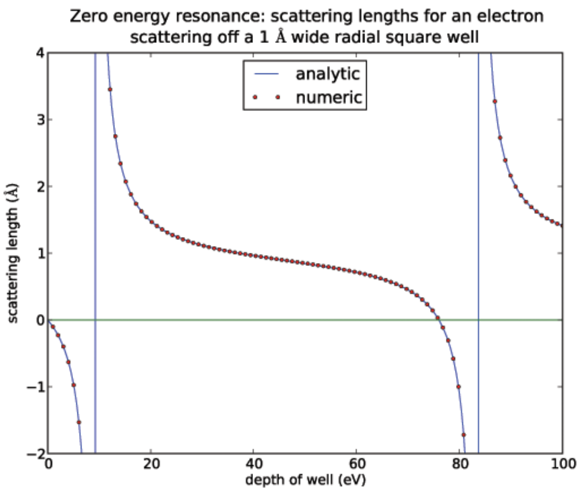


FIG. 3: (Left) Comparison of the analytic calculation of phase shifts for a radial square well with those generated by numerical methods. Parameters for this calculation were chosen to be experimentally realistic. The numerical and analytic results were found to agree to 3 or 4 significant figures when 10,000 integration steps were taken for the radial wavefunctions.

FIG. 4: (Right) Zero energy resonance in a radial square well. If the well has a bound state close to zero energy, or is close to taking one on, then scattering events can have very large scattering lengths. As the well takes on a new bound state the scattering length diverges. There is good agreement between the numerically calculated values and the analytic solution.

and the equation for $\tan\delta$ is:

$$\tan\delta_\ell(k) = \frac{k\, j'_\ell(ka) - \gamma_\ell(k) j_\ell(ka)}{k\, n'_\ell(ka) - \gamma_\ell(k) n_\ell(ka)}. \tag{14}$$

The logarithmic derivative $\gamma_\ell(a)$ for scattering off a radial square well of depth U_0 and width a_0 is given by the analytic expression (Bransden and Joachain 2005 eqn 12.111):

$$\gamma_\ell(k) = \frac{\kappa j'_\ell(\kappa a_0)}{j_\ell(\kappa a_0)}, \tag{15}$$

where $\kappa = \sqrt{k^2 + U_0}$, and so this provides a comparison against which the numerical solutions can be checked. An example of this is given in Fig 3.

C. Scattering Length and Zero Energy Resonance

Since I already had functions to calculate scattering phase shifts at a particular wavenumber for a given potential, a program to calculate scattering lengths for an arbitrary potential

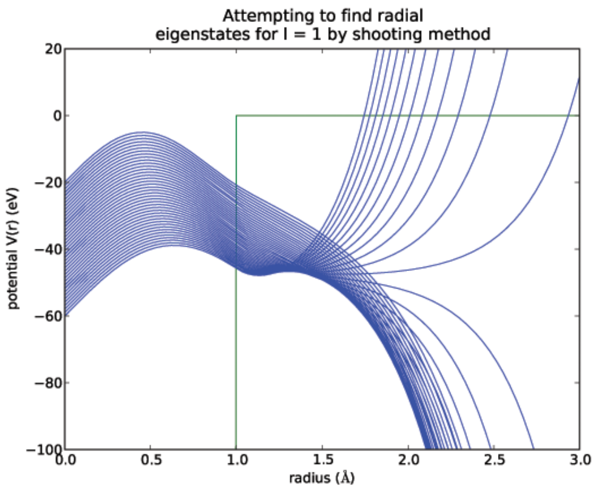


FIG. 5: The shooting method works by observing a flip in the divergent tail of the wavefunction as possible energies are scanned over. The energy for a bound state is assumed to lie between two energies with different signed tails.

needs simply to numerically evaluate the limit:

$$a_s = \lim_{k\to 0} \frac{\tan\delta_0}{k}, \tag{16}$$

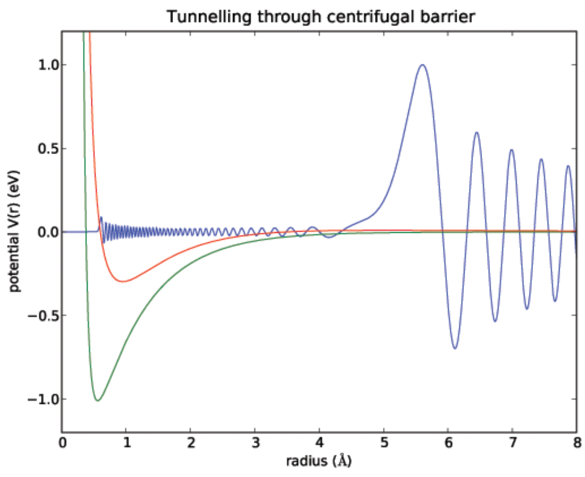
where δ_0 is the s wave phase shift and k the wavenumber of the scattering particle. This was done in Python simply by evaluating the above expression with a "small" k, such as 1 radian per metre, and decreasing k by powers of ten until convergence was reached with some tolerance. In practice the evaluation always converged immediately with such a starting value. This method was used to calculate the scattering lengths for electrons incident with a $1\AA$ wide radial square well (Fig. 4), as a function of well depth. For the radial square well the exact solution to this configuration is known and is [4]:

$$a_s = a_0\left(1 - \frac{tan(a_0\sqrt{U_0})}{a_0\sqrt{U_0}}\right), \tag{17}$$

where a_0 is the well's radius and $U_0 = \frac{2mV_0}{\hbar^2}$ is its reduced depth.

D. Bound States and Singular Potentials

In order to properly study Feshbach and zero energy resonances, I thought it would be a good idea to be able to say what the bound states of any particular well was. Although



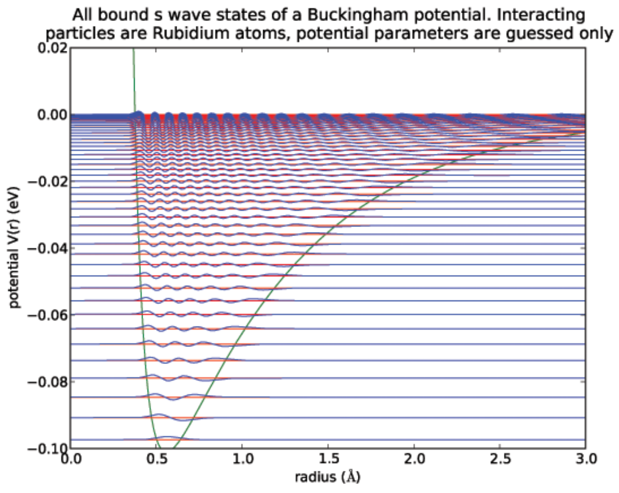


FIG. 6: (Left) A low energy free wavefunction encounters a Buckingham potential at a high angular momentum quantum number. The centrifugal barrier means that the wavefunction has to tunnel to get into the well - reducing somewhat the particle's probability of being found there. The green line (lower one if you're reading in black and white) is the actual potential, and the red one (upper) is the potential plus centrifugal term.

FIG. 7: (Right) All 46 $\ell = 0$ bound states of a particular Buckingham interatomic potential. The parameters for the potential are guessed only, but the Rubidium mass is used for the particles themselves.

I never ended up using it for anything of the sort, I write a program to find bound states of arbitrary potentials. The method employed was simply solving the radial Schrödinger equation (5) with negative energy, or equavalently, imaginary free space wavenumber k. To determine bound states, a range of imaginary wavenumbers were attempted, with the ones corresponding to bound states narrowed down using the shooting method [5]. Basically this method works by observing that for some energy, the tail of the wavefunction diverges off to positive infinity, say (this is how you know it's not a valid bound state), and for some other energy nearby, the tail goes off to negative infinity (Fig. 5). The energy at which the tail goes to zero for large r is assumed to lie between two such energies, and can be narrowed down by a search algorithm such as the golden ratio search [5] to arbitrary accuracy.

Realistic interatomic potentials are not radial square wells, and are generally singular at the origin. Therefore I modified my program for solving the radial Schrödinger equation to be able to handle singular potentials. The problem with singular potentials is that when numerical integration starts near the origin, the wavefunction finds itself out of its depth with the potential energy greater than its total energy, and so it does what any wavefunction with negative net energy does[3]: it exponentially grows.

To avoid overflow, I simply had the numerical integrator check for imminent overflow and prevent it by dividing the entire wavefunction calculated so far by some colossal number. This is acceptable since the entire wavefunction is only defined up to a multiplicative constant, and can be normalized later if need be. This enabled the calculation of bound and free (Figs 7 and 6) states for the Buckingham potential:

$$V(r) = -Ae^{-Br} + \frac{C}{r^6}. \tag{18}$$

IV. SIMULATION OF BOSE-EINSTEIN CONDENSATES

The aim of this part of the project was to simulate in 1D the collision of a free 'bullet' BEC with a 'target' BEC which is trapped by an optical potential (Fig 8). The idea is that a magic wavelength is chosen such that the target BEC sees the potential, but the bullet BEC does not. Simulations of this involved first setting up the initial conditions by finding the groundstate of the traps in which the BECs are prepared, one being the optical trap and the other a parabolic potential in which the bullet BEC is prepared. This potential represents the magnetic potential in which BECs are commonly prepared. In an experiment, the magnetic field can be moved to set the BEC in it moving as well, and then switched off - leaving the BEC moving in free space at the chosen speed.

The next step in the simulation was to advance the initial configuration in time using coupled Gross-Pitaevskii equations. Animations of these collisions were produced for the specific case in which both BECs are Rubidium 87 [4], and the reflection coefficient was determined at many different collision energies by running this simulation over a range of incident energies.

[3] Well, except when that exponential's coefficient has to be zero to satisfy some condition or other.

[4] Although this is not realistic with regard to one being blind to the trap - they will either both see the trap or both not, since they are the same species!

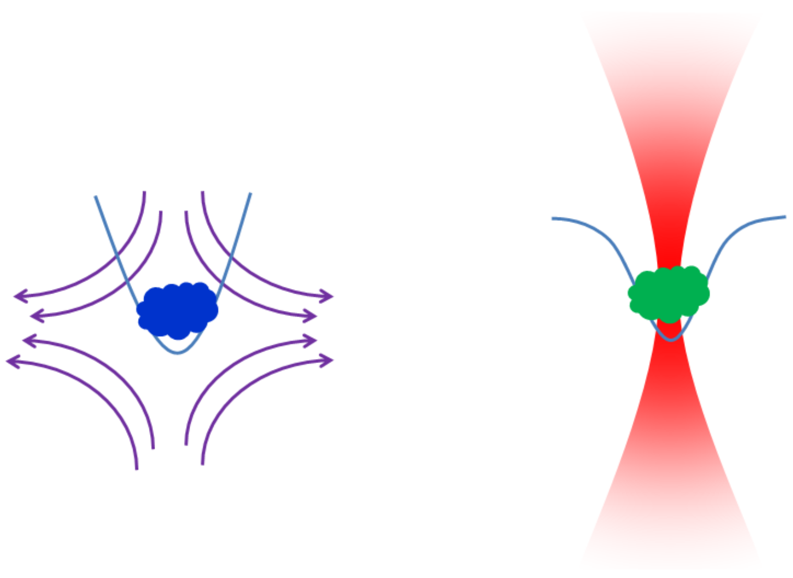


FIG. 8: The initial configuration of the two BECs. The bullet BEC is prepared in a parabolic trap which represents what would experimentally be a trapping magnetic field. The target BEC is prepared in a Gaussian trap, representing the optical dipole force at the focus of a laser. The system is then evolved in time, with the bullet BEC given some initial rightward momentum and the parabolic trap switched off.

A. XMDS

XMDS [6] (eXtensible Multi Dimensional Simulator) is an open source numerical integration package that takes input in XML, and writes C++ code to do the required simulation. Many thanks to Gary Ruben, who suggested it to me. It's what he uses for his BEC simulations [7], as do many condensed matter physicists around the world.

In the XML simulation scripts that XMDS takes as input, the user is required to write a high level description of the problem being solved, usually a partial differential equation. Details of the dimensions of the region of the simulation are entered, as well as the sequence of numerical integrations to perform. The user writes snippets of C++ code for the actual equations being solved, and XMDS inserts them into the correct loop structure in the resulting C++ code, using the numerical method of the user's choice.

The most commonly employed algorithm used is called the RK4IP method (Fourth Order Runge-Kutta in the Interaction Picture) [8]. Time propagation in this method uses the RK4 algorithm, and so are fourth order accurate. Spatial derivatives are evaluated using a method in which the field is first Fast Fourier Transformed into k space, then multiplied by ik once for each required derivative (the Fourier transform of a derivative is ik times the Fourier transform of the function), and then inverse transformed back. This gives the derivative of highest order accuracy possible for the number of x gridpoints used.

B. The Gross-Pitaevskii Equation

I wrote simulation scripts in XMDS to solve coupled Gross-Pitaevskii equations in 1D:

$$i\hbar\frac{d\psi_1}{dt} = \left[-\frac{\hbar^2}{2m_1}\frac{d^2}{dx^2} + g_{11}|\psi_1|^2 + g_{12}|\psi_2|^2\right]\psi_1, \tag{19}$$

and

$$i\hbar\frac{d\psi_2}{dt} = \left[-\frac{\hbar^2}{2m_2}\frac{d^2}{dx^2} + V(x)g_{22}|\psi_2|^2 + g_{21}|\psi_1|^2\right]\psi_2, \tag{20}$$

where Ψ_1 is the field of the bullet BEC, and Ψ_2 of the target. Only the target sees the potential $V(x)$, and both BECs experience their own self repulsion[5] as well as repulsion from each other. The coefficients g_{ij} determine the strength of these repulsions and are given by:

$$g_{ij} = \frac{4\pi\hbar^2 a_{ij}}{m_i}, \tag{21}$$

where a_{ij} is the s wave scattering length between the ith and jth atomic species in the condensate.

Reducing the GPE to 1D is not as trivial as for the Schrödinger equation. You might notice that the above equations would have mismatched units if the particle densities $|\Psi_1|^2$ and $|\Psi_2|^2$ were in atoms per unit length. Due to the nonlinearity, our densities must still be in atoms per unit volume, but we will consider the fields to be constant in two spatial dimensions and varying only in the x direction. Thus this situation describes the a collision of two infinite parallel planes of BEC.

Because of this, we can no longer speak of a total number of particles (it would be infinite). Instead, what we get when we integrate the particle densities over the x direction is the number of atoms per unit area of this plane. This quantity is called a 'column density', and is our normalization condition.

But before evolving coupled equations in time, an XMDS program was used calculate the initial conditions.

[5] Interactions can be attractive, but only repulsive ones are considered for these simulations.

C. Groundstate calculation

The groundstates of both traps were determined using a relaxation method known as imaginary time evolution [9]. The method works by advancing an initial guess in imaginary time, and is guaranteed to decrease its energy at every timestep. After sufficient (imaginary) time, the energy can decrease no more and has converged to that of the groundstate.

Consider the stationary solutions to the Gross-Pitaevskii equation for the order parameter field Ψ:

$$\Psi(x,t) = \psi(x)e^{-\frac{i\mu}{\hbar}t}, \tag{22}$$

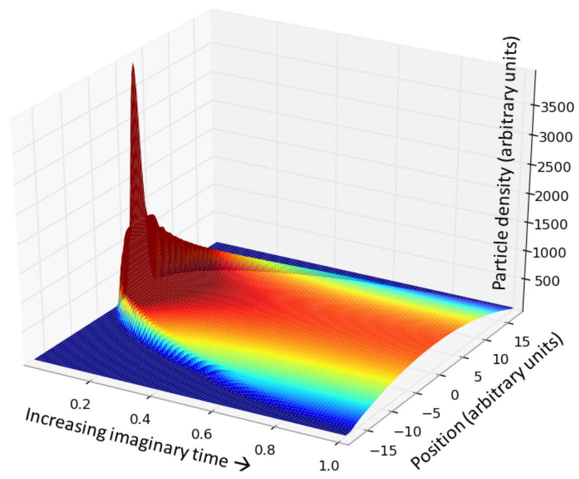
where μ is the chemical potential[6]. These solutions are oscillatory, but if we make the substitution $t \rightarrow -i\tau$, they become:

$$\Psi(x,\tau) = \psi(x)e^{-\frac{\mu}{\hbar}\tau} \tag{23}$$

which is exponential, either growing or decaying. If it is growing, then the ground state, with the smallest chemical potential, will grow faster than all other states. If it is decaying, the ground state will decay the slowest. Propagation in imaginary time does just that, propagating the GPE with this substitution, and renormalizing at every step to maintain the correct number of particles. After sufficient (imaginary) time, only the ground state is left (Fig. 9). This can then be output to a file and used as the initial conditions for the ordinary propagation in time.

To test how accurate these groundstate calculations were, they were advanced in time with both the potentials switched on, and plots of the particle density viewed to verify that it was unchanging. This appeared to be the case. However, when energy 'measurements' were taken (as described in the next section), It was observed that both BECs were still in a slightly excited state. For the target BEC (which was in the narrower well), the observed oscillations in self and potential energy were on the order of a tenth a percent of those energies themselves. This small error is a 'breathing mode' in which the cloud of atoms expands and contracts at a quantized frequency.

[6] Analogous to the total energy for Schrödinger wavefunctions, but including a self energy due to the nonlinear self repulsion.



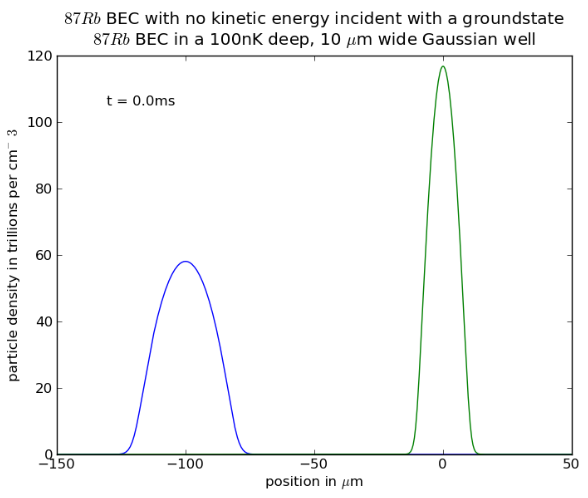


FIG. 9: (Left) A tall Gaussian initial guess of the order parameter is evolved in imaginary time to find the ground state of a harmonic potential. It decays or grows at every integration step, but is renormalized constantly to ensure the correct number of atoms remains.

FIG. 10: (Right) An example of initial conditions determined by imaginary time propagation. The bullet BEC (left one) was prepared in a 100 rad per sec harmonic trap, and the target (right) in a Gaussian well with depth 100nK and standard deviation 10 microns.

Whilst the energy oscillations were small compared to the energies themselves, they were on the order of 10% of the energy exchanges observed in the lower energy collisions. Increasing the length of relaxation (imaginary) time in preparing the groundstates did not make these oscillations smaller. Rather, they only became smaller if the relaxation stepsize was decreased. This seemed strange, as relaxation methods usually converge eventually regardless of the stepsize - in fact they usually converge faster for larger stepsizes.

A little thought on the matter revealed that the culprit was the use of the fourth order Runge-Kutta method for the integration steps in imaginary time. As illustrated by Fig 11, when an integration step is taken, the field exponentially grows or decays (Lets use growth to illustrate the point, but the same argument applies for decay). After the integration step, the field has a normalization that is too large, and is subsequently bumped down to the correct value before the next step.

It should be noted that, if the algorithm took more integration steps before normalizing, the resulting field would be more spread out than desired. The field with larger norm has more self repulsion than the one with the correct norm, and hence taking steps toward *its* groundstate will give a groundstate that is more spread out than the one we seek.

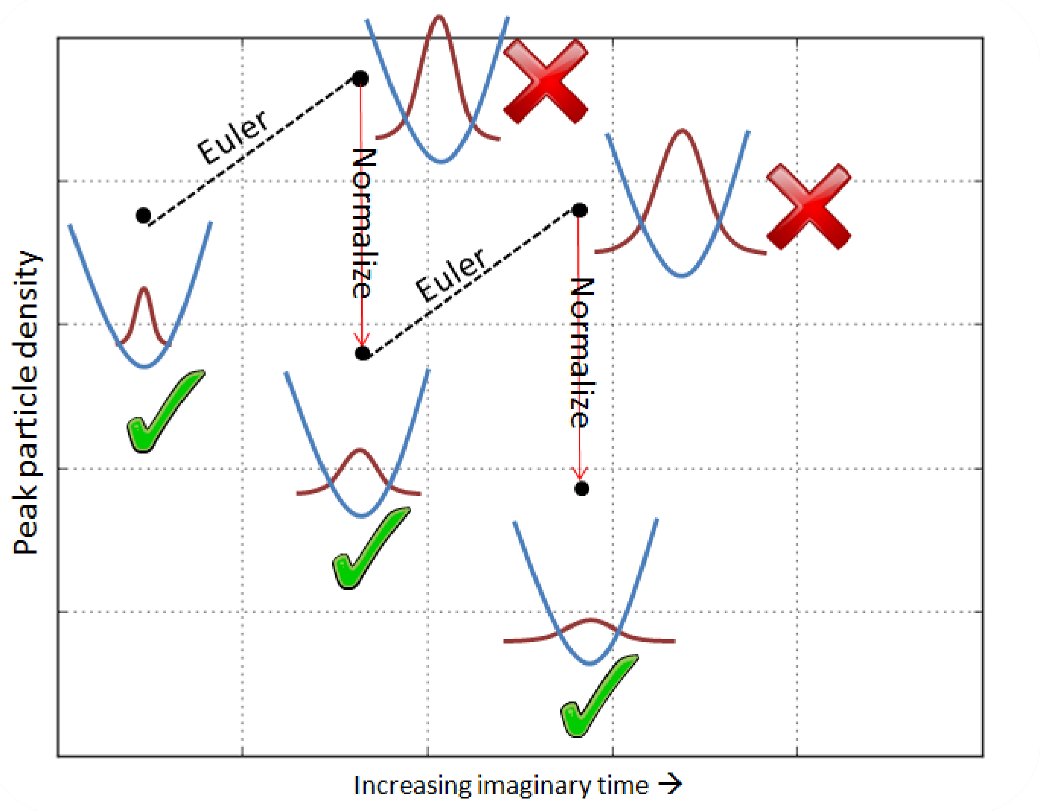


FIG. 11: Diagram of the Euler method evolving the $x = 0$ point of an initial guess in imaginary time, with normalization imposed at each step. The shape of the entire field is shown at each step for illustrative purposes. After an integration step, the field is in a state that does not satisfy normalization. Higher order methods step in time using the state of the field at an intermediate point, which also does not satisfy normalization. These methods do not converge for this reason, unless prohibitively small step sizes are taken.

High order methods such as the fourth order Runge-Kutta integrate not only based on the state of the field at the start of the step, but also on the state of it at the end. The slope they use to take a step is actually one belonging to a state of the field somewhere in the middle of the step. In the imaginary time evolution method, *this state has too great a normalization*. Therefore taking steps using it yields groundstates that are too spread out. This would give rise to the breathing mode mentioned earlier, and shown in Fig 12.

The only solution, if one insists on the RK4 method, is to take very small step sizes, as to reduce how far from correct normalization these states are. This is computationally expensive, and there is a much better way. Replacing the RK4 method with the Euler method ensures that the system is only ever evolved based on the state of the field at the beginning of an integration step. This allows one to take step sizes as large as don't cause overflow

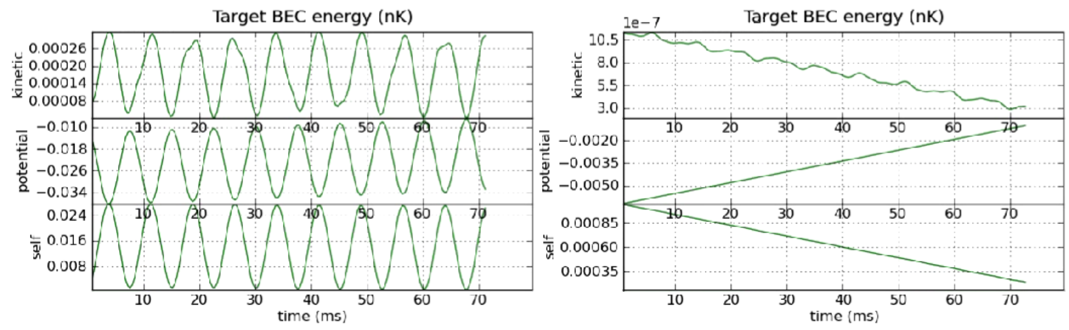


FIG. 12: The left set of plots is the energy per atom of the target BEC after a groundstate calculation using the RK4 method in (imaginary) time. The right set used the euler method, and step sizes 100 times larger. Both are being evolved in ordinary time here. The y axis labels are changes in energy from an offset rather than absolute energy. The kinetic energy plot on the right has its y axis units multiplied by an exponent written above it. The left plot clearly shows a breathing mode, and so is dominated by the error in the groundstate calculation. The right plots are dominated instead by the much smaller error due to the numerical integration in ordinary time.

(in my case a few hundred times larger) and gives correct convergence to the groundstate. Combined with the speedup of a lower order method, the groundstate calculation for this project ran in seconds, as opposed to an hour before making the change. XMDS does not include any methods which are Euler in time, however one is easily implemented with the freedom it allows the user to manipulate the fields at any point in the simulation.

I'm unsure of whether this is an original idea or not, as the example script written by Joe Hope that comes with XMDS uses the RK4IP method for a groundstate calculation, as does Gary Ruben [7] (though he simply adopted the example script for his own purposes, as did I). Yang and Lakoba, who have developed accelerated versions of the imaginary time method [10], write that the Euler method is "the simplest implementation", but I haven't seen anyone state that it is in fact the best as well.

D. Time Evolution

Once the groundstates of both wells were calculated, they were used as input for another XMDS simulation that would evolve them in time, according to the coupled Gross-Pitaevskii equations (19) and (20). But first the field of the bullet, Ψ_1 had to be given some rightward momentum, physically corresponding to it having been in a moving trap. To this end it was multiplied through by a plane wave of wavenumber:

$$k = \frac{\sqrt{2m_1 E_k}}{\hbar}, \tag{24}$$

where E_k is the desired kinetic energy for the simulation. k was included as a command line argument for the simulation code, enabling something like a Python script to call the simulation to run over a range of k values. The time evolution then used the RK4IP method to evolve the initial conditions for a set amount of time (a few tens of milliseconds). Whilst the time evolution stepsize was quite small, the field was only sampled once every 200 timesteps or so. Several quantities calculated from this sampling were output at the end of the calculation, including:

- Particle densities
- Spatial Fourier transforms of the fields
- Energy of the fields, in all forms:
  - Kinetic
  - Potential (of the target)
  - Self (due to self repulsion)
  - Coupling (due to mutual repulsion)
- Leftmoving and rightmoving proportions of bullet BEC

The energy densities of the bullet and target BECs are, respectively[7] [11]:

$$\epsilon_1 = \frac{\hbar^2}{2m_1}|\nabla\Psi_1|^2 + \frac{1}{2}g_{11}|\Psi_1|^4 + \frac{1}{2}g_{12}|\Psi_1|^2|\Psi_2|^2, \tag{25}$$

and

$$\epsilon_2 = \frac{\hbar^2}{2m_2}|\nabla\Psi_2|^2 + \frac{1}{2}g_{22}|\Psi_2|^4 + \frac{1}{2}g_{21}|\Psi_2|^2|\Psi_1|^2, \tag{26}$$

[7] I added in the coupling energy terms myself, and they are probably different if the two atomic species are not the same, due to the reduced mass not being half of the individual masses in that case.

in which the separate terms give, in order, the kinetic, self, coupling, and potential (for the target) energies. At each point in time that the field was sampled, these energy densities were integrated over all x, reducing them from energy per unit volume to energy per unit area[8]. Dividing this by the normalization, which is in atoms per unit area, gives the energy per atom for each of these forms of energy.

The left moving and right moving proportions were used to determine the reflection coefficient of each simulation. In Fourier space, the field of the bullet was split into two fields, one with positive wavenumber and the other with negative. Back in real space, integrating the densities of these fields gave the proportions of left/right moving BEC. This method was more effective than simply waiting for the interaction to finish and seeing how much BEC was on each side of the region. This was because the periodic boundary conditions imposed by the RK4IP method allowed BEC to pass from one side of the region to the other. Thus basing the reflection coefficient on *where* the BEC was had more room for error than looking at *what direction* it was traveling in.

With all these 'measurements', the simulation was run over a range of kinetic energies, with the following parameters:

- Normalization for both BECs was chosen such that the infinite planes had a million atoms in an area of $2\pi\sigma^2$ with $\sigma = 10\ \mu$m. So the 1 dimensional normalization condition was $N_{1\mathrm{D}} = 1.6 \times 10^{11}\mathrm{cm}^{-2}$.
- Both BECs had the atomic mass and scattering length of Rubidium 87.
- The harmonic trap in which the bullet BEC was prepared had a frequency of $\omega = 20$ rad sec^{-1}.
- The Gaussian trap had standard deviation of $\sigma = 10\mu$m, and a depth of $V_0 = 290$ nK (4×10^{-30}J).
- The bullet BEC's original position was 100μm to the left of the target.

A Python script was used to run all these simulations in succession, and it produced videos of the states of the fields evolving over time. The reflection coefficient of each interaction

[8] Remember we're considering infinite planes of BEC, so all integrals over x give quantities per unit area of these planes.

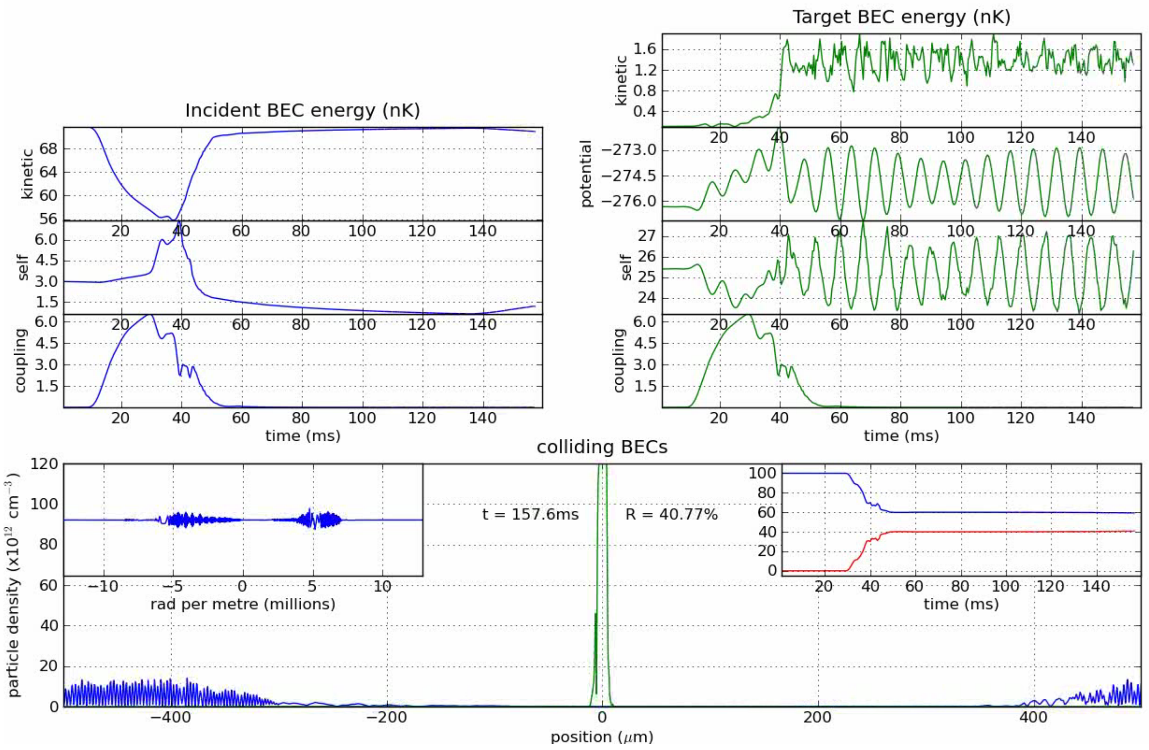


FIG. 13: The final state of a simulation with intermediate energy. Note that due to the periodic boundary conditions, the reflected and transmitted components are merging over the boundary. This is unphysical and the reflection coefficient was read off all simulations before this interaction.

could be read off these videos as the proportion of the bullet BEC which was leftmoving after the collision.

V. RESULTS AND DISCUSSION

The final state of a typical simulation is shown in Fig 13. One visible feature is a drop in the bullet's kinetic energy as it encounters the target, and a corresponding rise in the coupling energy of both BECs. The target BEC is left in a breathing mode, in which its energy oscillates between self energy and potential energy, and they appears to be a beat note in these oscillations, perhaps due to multiple modes being excited. Also, the kinetic energy of the target is left varying about a mean value, but not with an obvious periodicity. A few frames of the interaction itself are shown in Fig 14. Most collisions had cusp-like features, where the particle density of the bullet BEC became very large during a reflection. Peaks like this are a general feature of reflections in quantum mechanics [12], and can be intuitively imagined as the result of a stream of classical particles slowing, stopping, and

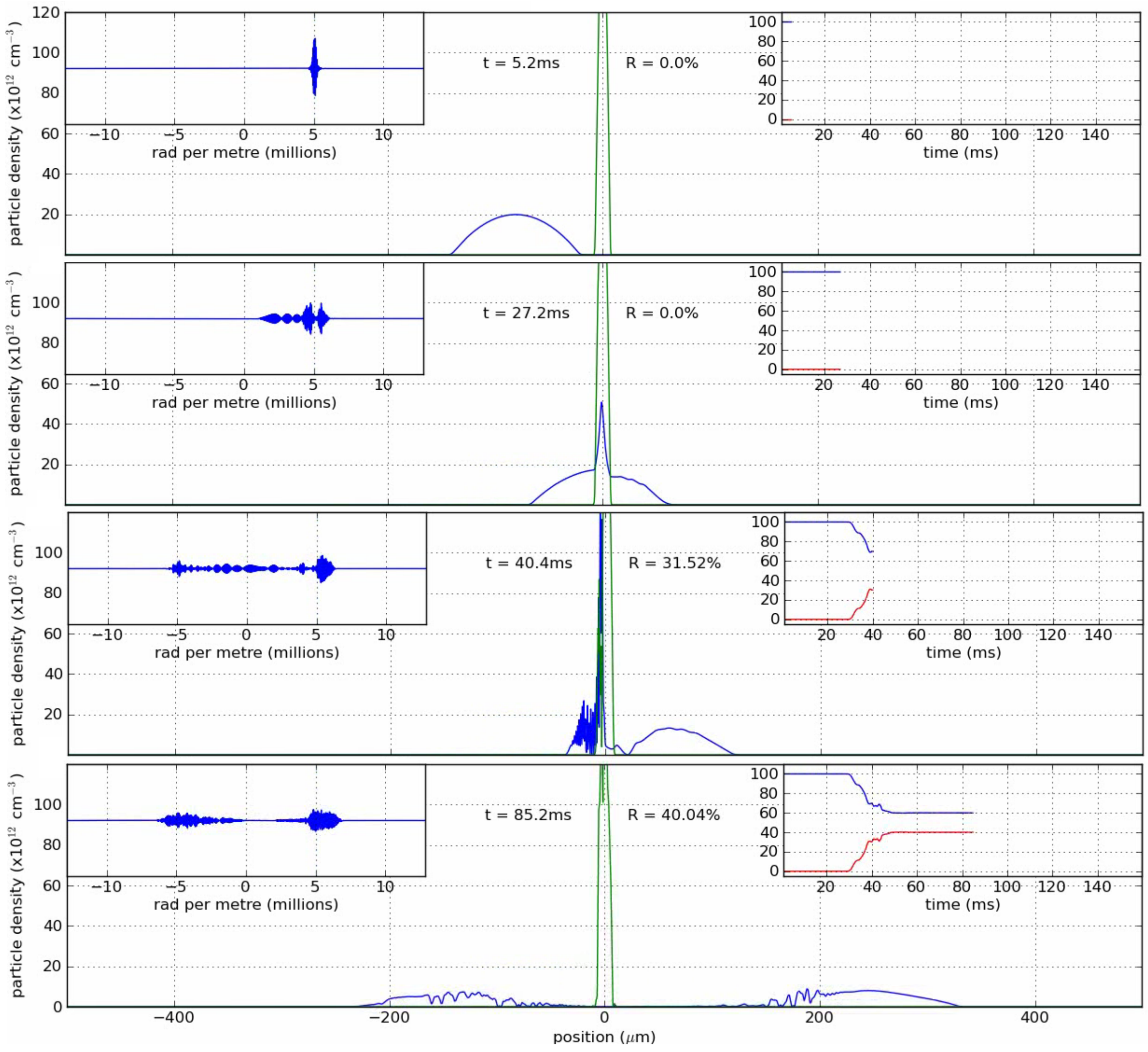


FIG. 14: Several frames from the time evolution of the same simulation as in Fig 13. There is an increased density in the part of the bullet BEC overlapping the target before a cusp-like peak forms and some of the bullet is reflected. Note that the formation of the peak corresponds to the peak in the bullets self energy seen in Fig 13. The upper left plot in each frame is the bullet's spatial Fourier transform, and can be seen to have a leftmoving component after the collision. The upper right plot is the percentage left and rightmoving, with the red line being leftmoving. The percentage leftmoving is shown on the plot as 'R= . . . %'. It stabilizes after the collision to a number which is then read off as the reflection coefficient for this energy.

turning around. Their density will be inversely proportional to their velocity, hence at a classical turning point, the particle density will diverge[9]. Supporting this idea is the fact that these peaks occurred when the Fourier transform of the bullet BEC first obtained negative

[9] A difference being, of course, that in QM the density doesn't actually become infinite

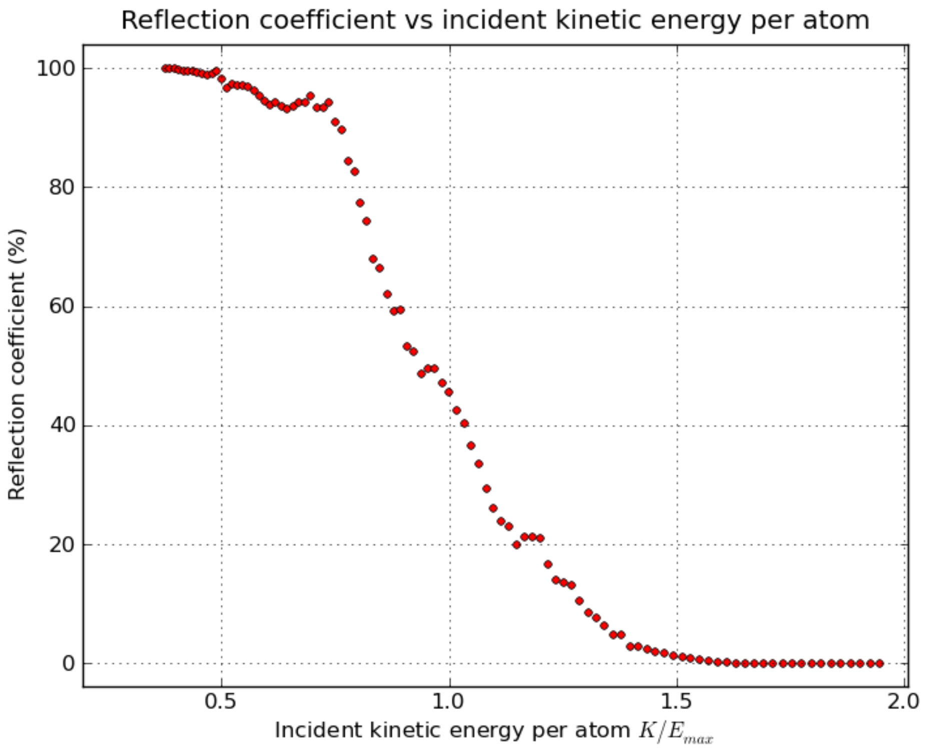


FIG. 15: The reflection coefficient of the interaction, as a function of incident kinetic energy per atom. There are several resonances, at which there is a local peak in the reflection coefficient. These resonances appear to be roughly equally spaced, and are attributed to the energy exchange between the two BECs being greatest when the target is excited into a discrete mode.

wavenumbers, indicating that a component of the field had turned around is was moving leftward.

Plotting the reflection coefficient for 100 runs of this simulation over a range of energies gave Fig. 16. Since the barrier that the bullet encounters is due to its coupling energy with the target, the maximum barrier height it encounters is:

$$E_{\max} = g_{12}|\Psi_2|^2_{\max}. \tag{27}$$

$|\Psi_1|^2_{max}$ in this run of simulations was 1.73×10^{14} cm^{-3} at $t = 0$, and whilst it changed during the collision, this is the value used for calculating $E_{\max}$. Writing the incident kinetic energies as a proportion of the maximum barrier energy allows easier comparison with Schrödinger scattering.

One difference is the fact that the reflection coefficient at $K/E_{\max} = 1$ is not 50% as it would be in the Schrödinger case. It is lower, at about 45%. One source of kinetic energy for the bullet BEC is its self energy. As a BEC moves in free space, it spreads out due to self repulsion, and its self energy is transformed into kinetic. So the bullet BEC could have gained some kinetic energy here[10], allowing more of it to be transmitted.

Another possible explanation would be that the use of a constant E_{max} was unwarranted, as the central peak of the target would be destroyed upon impact.

However, the energy exchange between bullet and target BEC is in the target's favour[11]. So this should decrease the bullet BEC's transmission, since it's given up some energy to the barrier, and thus made it higher.

The unwarranted use of a constant E_{max} is probably the culprit, and a time averaged E_{max} could be taken to resolve this in future.

These same considerations of energy exchange can be used to explain the observed resonances in Fig (16). The Target BEC has discrete excited states that it can be partially excited to, and so when the collision energy matches the energy required to excite atoms to such a state, there is a larger energy transfer. This leaves the bullet BEC with less energy than when off resonance, causing an increase in the proportion reflected.

VI. CONCLUSION

The existence of magic wavelengths allows for experiments in which one BEC is held in an optical trap and another is not. Scattering of the free BEC off the trapped one has many similarities with Schrödinger scattering, and several differences. The fact that the incident BEC can interact with the barrier it faces gives rise to an energy exchange, whereby the trapped BEC gains energy. Resonances in this interaction give rise to local peaks in the reflection coefficient at certain energies.

Using XMDS to simulate BEC dynamics is very effective, allowing for fast simulations without spending a lot of time writing code. The RK4IP algorithm allows for accurate time propagation of the system, and the imaginary time evolution method is an easily

[10] The frequency of the harmonic trap was chosen as to "prespread" the bullet BEC and minimise this effect, however, and the energy plots show that there is not much of it pre-collision. This was done to prevent some of the bullet BEC moving leftward before the collision itself, which would make reflection 'measurements' meaningless.

[11] It has to be - the targets energy can't get any lower than the groundstate!

implemented method for finding groundstates of BECs. Using the Euler method to take integration steps in imaginary time is vastly superior to using higher order methods, and gives a substantial speedup.

Having learnt how to run simulations of this type, and having shown that BEC scattering has an energy dependence, I hope in the future to extend these simulations to scanning over different parameter spaces, in higher dimensions, or simulating the behaviour of BECs in magic wavelength optical lattices. Such simulations may shed light on phenomena such as superfluidity/superconductivity in condensed matter systems, and help motivate experiments to be carried out to do the same.

Acknowledgments

Thanks to my supervisors Kristian and Lincoln for guiding me through this project with helpful suggestions and ideas. Thanks to Gary Ruben for his help with XMDS, and to my friends and fellow students for helpful and interesting discussions. I would also like to acknowledge the many members of the free and open source software community, for developing quality software packages without which work like this would be much more difficult.

---

[1] C. Chin, R. Grimm, P. Julienne, and E. Tiesinga, *Feshbach resonances in ultracold gases* (2008), URL `arXiv.org:0812.1496`.

[2] C. Foot, *Atomic Physics* (Oxford University Press, 2005).

[3] B. Bransden and C. Joachain, *Introduction to Quantum Mechanics* (Longman Scientific and Technical, 1989).

[4] B. Bransden and C. Joachain, *Physics of Atoms and Molecules* (Prentice Hall, 2003), 2nd ed.

[5] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. T. Flannery, *Numerical Recipes* (Cambridge University Press, 2007), 3rd ed.

[6] P. T. Cochrane, G. Collecutt, P. D. Drummond, and J. J. Hope, *Xmds documentation* (2008), URL `www.xmds.org`.

[7] G. Ruben, D. M. Paganin, and M. J. Morgan, *Vortex-lattice formation and melting in a*

nonrotating bose-einstein condensate (2008), URL `arXiv.org:0801.0383`.

[8] B. M. Caradoc-Davies, Ph.D. thesis, Univ. Otago, Dunedin, New Zealand (2000).

[9] L. Lehtovaara, J. Toivanen, and J. Eloranta, J. Comput. Phys. **221**, 148 (2007), ISSN 0021-9991.

[10] J. Yang and T. I. Lakoba, *Accelerated imaginary-time evolution methods for the computation of solitary waves* (2007), URL `arXiv.org:0711.3434`.

[11] C. J. Pethick and H. Smith, *Bose-Einstein Condensation in Dilute Gases* (Cambridge University Press, 2002).

[12] M. V. Berry and N. L. Balazs, American Journal of Physics **47**, 264 (1979).

Appendix: simulation source code

The three files used for the BEC simulations are included in this appendix. They are:

- initial.xmds - Initializes both BECs to the groundstates of their respective traps, using the imaginary time evolution method.
- coupled.xmds - Propagates the initial conditions in time, and samples the particle densities and energies of the BECs
- run.py - runs initializations as required, then runs 'initial' over a range of energies, and produces a video of each of these simulations.

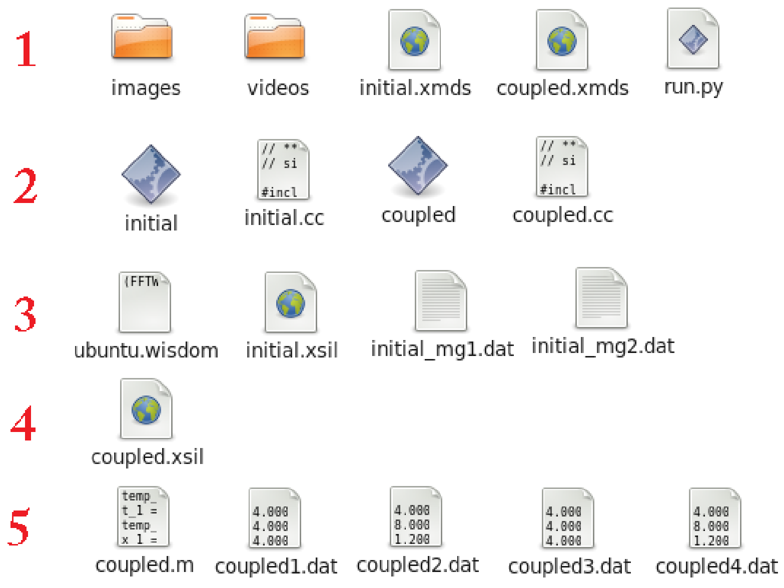


FIG. 16: 1. The three files must be in the same directory, along with two empty folders that will later be filled with images and videos. All the other files pictured are created when 'run.py' executes.
2. When 'run.py' is executed, it first runs XMDS on the two '.xmds' files to generate and compile C++ code, resulting in two .cc files, and two executables being created.
3. Then it executes 'initial' to do the groundstate calculations. The '.wisdom' file is 'wisdom' to speed up future runs. A '.xsil' file is generated that contains the results of the simulation. It is linked to the two data files which contain the actual field data.
4. 'run.py' then executes 'coupled'. This results in a '.xsil' file containing the results of the simulation. There are no other data files since the output format is ASCII.
5. 'run.py' then uses a utility to convert the '.xsil' file to plain text files. These are the four data files. The '.m' file is a Matlab script to import that data into matlab, which is automatically generated by the utility. 'run.py' then reads in the text files, makes plots (storing them temporarily in the images folder), and then generates an AVI video for each simulation. The videos are saved into the videos folder.

initial.xmds

```
<?xml version="1.0"?>

<!--                                                                  -->
<!--  This program is free software; you can redistribute it and/or    -->
<!--  modify it under the terms of the GNU General Public License      -->
<!--  as published by the Free Software Foundation; either version 2   -->
<!--  of the License, or (at your option) any later version.           -->
<!--                                                                  -->
<!--  This program is distributed in the hope that it will be useful,  -->
<!--  but WITHOUT ANY WARRANTY; without even the implied warranty of   -->
<!--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the    -->
<!--  GNU General Public License for more details.                     -->
<!--                                                                  -->
<!--  You should have received a copy of the GNU General Public License -->
<!--  along with this program; if not, write to the Free Software      -->
<!--  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston,           -->
<!--  MA  02110-1301, USA.                                              -->

<simulation>
  <name>initial</name>
  <author>Chris Billington</author>
  <description>
    Evolves two Gross-Pitaevskii equations in imaginary time to find the ground
    state of a harmonic trap, and a Gaussian trap. The Euler method is used
    rather than a higher order method, in order to be able to take larger
    steps without introducting an error inherent in higher order methods. This
    error is due to the fact that the assumptions made by higher order
    methods are not valid for this kind of calculation with renormalization
    at every step - to get convergence with higher order methods one must take
    a prohibitively small step size. The Euler method therefore gives a
    significant speedup, and can be demonstrated to converge better than
    higher order methods. This script based on the "groundstate.xmds" example
    script by Joseph Hope.
  </description>

  <prop_dim>t</prop_dim>
  <error_check>no</error_check>
  <stochastic>no</stochastic>
  <use_wisdom>yes</use_wisdom>
  <benchmark>yes</benchmark>

  <!-- Global variables for the simulation -->
  <globals>
    <![CDATA[
      //All in SI units
      const double m1 = 86.909181*1.660539e-27;
      const double m2 = 86.909181*1.660539e-27;
      const double a1 = 53.1e-10;
      const double a2 = 53.1e-10;
      const double N1 = 1.0e6;
      const double N2 = 1.0e6;
      const double sigma1 = 10.0e-6;
      const double sigma2 = 10.0e-6;
      const double V_0 = 4.00e-30;
      const double x0 = -100.0e-6;
      const double llim = -500.0e-6;
      const double rlim =  500.0e-6;
      const double stepsize = 1.0e-5;
      const double hbar = 1.055e-34;
      const double g1 = 4*M_PI*hbar*hbar*a1/m1;
      const double g2 = 4*M_PI*hbar*hbar*a2/m2;
      const double N1_1D = N1/(2*M_PI*sigma1*sigma1);
      const double N2_1D = N2/(2*M_PI*sigma2*sigma2);
      const double omega = 20;
    ]]>
  </globals>
```

```
<!-- Field to be integrated over -->
<field>
  <name>main</name>
  <dimensions> x </dimensions>
  <lattice>  4096  </lattice>
  <domains>  (llim,rlim) </domains>
  <samples>1 0</samples>

  <vector>
    <name>main</name>
    <type>complex</type>
    <components>psi1 psi2</components>
    <fourier_space>no</fourier_space>
    <![CDATA[
        //Initial guesses
        psi1 = sqrt(N1)*pow(2*M_PI*sigma1*sigma1,-0.75) *
               exp(-(x-x0)*(x-x0)/(4*sigma1*sigma1));
        psi2 = sqrt(N2)*pow(2*M_PI*sigma2*sigma2,-0.75)*exp(-x*x/(4*sigma2*sigma2));
    ]]>
  </vector>

  <vector>
    <name>potential</name>
        <type>double</type>
        <components>V1 V2</components>
        <fourier_space>no</fourier_space>
        <![CDATA[
            V1 = 0.5*m1*omega*omega*(x-x0)*(x-x0);
            V2 = -V_0*exp(-x*x/(2*sigma2*sigma2));
    ]]>
  </vector>

  <vector>
    <!-- these are the spatial derivatives, to be evaluated in k space -->
    <name>derivs</name>
    <type>complex</type>
    <components>deriv1 deriv2</components>
    <fourier_space>yes</fourier_space>
    <![CDATA[
        deriv1 = 0;
        deriv2 = 0;
    ]]>
  </vector>
</field>

<!-- The sequence of integrations to perform -->
<sequence>
  <!-- Outer loop that allows a sample of the field every 100 time steps -->
  <sequence>
    <cycles>100</cycles>
    <!-- The inner loop of integrate/renormalize cycles. No sampling. -->
    <sequence>
      <cycles>100</cycles>

      <!-- Evaluating the transverse derivatives in fourier space-->
      <filter>
        <vectors>main derivs</vectors>
        <fourier_space>yes</fourier_space>
        <![CDATA[
            deriv1 = -kx*kx*hbar/(2*m1)*psi1;
            deriv2 = -kx*kx*hbar/(2*m2)*psi2;
        ]]>
      </filter>

      <!-- Moving forward in time with the Euler method -->
      <filter>
        <vectors>main derivs potential</vectors>
        <fourier_space>no</fourier_space>
        <![CDATA[
            psi1 += stepsize*(deriv1 - 1/hbar * (V1 + g1*mod2(psi1))*psi1);
            psi2 += stepsize*(deriv2 - 1/hbar * (V2 + g2*mod2(psi2))*psi2);
        ]]>
      </filter>

      <!-- Renormalizing -->
      <filter>
        <moment_group>
          <moments>ncalc1 ncalc2</moments>
          <integrate_dimension>yes</integrate_dimension>
                <![CDATA[
                    //calculating the norm as it stands
                    ncalc1 += mod2(psi1);
                    ncalc2 += mod2(psi2);
                 ]]>
        </moment_group>
        <vectors>main</vectors>
        <fourier_space>no</fourier_space>
        <![CDATA[
            //bumping down to the correct norm
            psi1 *= sqrt(N1_1D/ncalc1.re);
            psi2 *= sqrt(N2_1D/ncalc2.re);
        ]]>
      </filter>
    </sequence>

    <!-- Sampling every 100 time steps, mainly for debugging -->
    <integrate>
      <algorithm>RK4IP</algorithm>
      <interval>stepsize</interval>
      <lattice>1</lattice>
      <samples>1 0</samples>
      <vectors>main</vectors>
        <![CDATA[//do nothing]]>
    </integrate>
  </sequence>

  <!-- A final sample of the whole field -->
  <integrate>
    <algorithm>RK4IP</algorithm>
    <interval>stepsize</interval>
    <lattice>1</lattice>
    <samples>0 1</samples>
    <vectors>main</vectors>
      <![CDATA[//do nothing]]>
  </integrate>
</sequence>

<output format = "binary" precision = "double">
  <!-- This output for visual checks and debugging -->
  <group>
    <sampling>
          <fourier_space> no </fourier_space>
          <lattice> 512 </lattice>
          <moments>density1 density2</moments>
          <type>double</type>
          <![CDATA[
              density1 = mod2(psi1);
              density2 = mod2(psi2);
          ]]>
    </sampling>
  </group>

  <!-- actual solution output -->
  <group>
    <sampling>
          <fourier_space> no </fourier_space>
          <lattice> 4096 </lattice>
          <moments>psi1R psi1I psi2R psi2I</moments>
          <type>double</type>
          <![CDATA[
              psi1R=psi1.re;
              psi1I=psi1.im;
              psi2R=psi2.re;
              psi2I=psi2.im;
          ]]>
    </sampling>
  </group>
</output>
</simulation>
```

coupled.xmds

```
<?xml version="1.0"?>

<!--                                                                   -->
<!--   This program is free software; you can redistribute it and/or     -->
<!--   modify it under the terms of the GNU General Public License       -->
<!--   as published by the Free Software Foundation; either version 2    -->
<!--   of the License, or (at your option) any later version.            -->
<!--                                                                   -->
<!--   This program is distributed in the hope that it will be useful,   -->
<!--   but WITHOUT ANY WARRANTY; without even the implied warranty of    -->
<!--   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the     -->
<!--   GNU General Public License for more details.                      -->
<!--                                                                   -->
<!--   You should have received a copy of the GNU General Public License -->
<!--   along with this program; if not, write to the Free Software       -->
<!--   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston,            -->
<!--   MA  02110-1301, USA.                                               -->

<simulation>
  <name>coupled</name>
  <author>Chris Billington</author>
  <description>
    Evolves the Gross-Pitaevskii equation for a coupled system of two BECs
    in one spatial dimension. Two groundstate BECs are read in from the
    "initial.xsil" file, the first being the groundstate of a harmonic
    potential and the second of a Gaussian potential. The first order
    parameter field is multiplied by a plane wave of a specified wavelength
    to set it moving rightward, and it is evolved in time without seeing any
    potential (as if its magnetic trap was turned off whilst moving). The
    second BEC remains in its Gaussian trap, but the two are allowed to
    interact via a coupling term in the equations.
  </description>

  <prop_dim>t</prop_dim>
  <error_check>no</error_check>
  <stochastic>no</stochastic>
  <use_wisdom>yes</use_wisdom>
  <benchmark>yes</benchmark>

  <!-- Global variables for the simulation -->
  <argv>
    <arg>
      <!-- a command line argument - is called with many values by a python script -->
      <name> k </name>
      <type> double </type>
      <default_value> 6.0e6 </default_value>
    </arg>
  </argv>
  <globals>
    <![CDATA[
    const double m1 = 86.909181*1.660539e-27;
    const double m2 = 86.909181*1.660539e-27;
    const double a1 = 53.1e-10;
    const double a2 = 53.1e-10;
    const double a3 = 53.1e-10;
    const double N1 = 1.0e6;
    const double N2 = 1.0e6;
    const double sigma1 = 10.0e-6;
    const double sigma2 = 10.0e-6;
    const double V_0 = 4.00e-30;
    //const double k = 6.0e6; (a command line argument at the moment)
    const double llim = -500.0e-6;
    const double rlim = 500.0e-6;
    const double hbar = 1.055e-34;
    const double N1_1D = N1/(2*M_PI*sigma1*sigma1);
    const double N2_1D = N2/(2*M_PI*sigma2*sigma2);
    const double g1 = 4*M_PI*hbar*hbar*a1/m1;
    const double g2 = 4*M_PI*hbar*hbar*a2/m2;
    const double gc1 = 4*M_PI*hbar*hbar*a3/m1;
    const double gc2 = 4*M_PI*hbar*hbar*a3/m2;
  ]]>
</globals>

<!-- Field to be integrated over -->
<field>
  <name>main</name>
  <dimensions> x </dimensions>
  <lattice> 4096  </lattice>
  <domains> (llim,rlim) </domains>
  <samples>0 0 0 0</samples>

  <vector>
    <name>main</name>
    <type>complex</type>
    <components>psi1 psi2</components>
    <!-- importing data from previously done initialization -->
    <filename format = "xsil" moment_group = "2" geometry_matching_mode = "loose">
      initial.xsil
    </filename>
    <fourier_space>no</fourier_space>
  </vector>

  <vector>
    <name>potential</name>
    <type>double</type>
    <components>V</components>
    <fourier_space>no</fourier_space>
    <![CDATA[
        V = -V_0*exp(-x*x/(2*sigma2*sigma2));
    ]]>
  </vector>

  <vector>
    <!-- These spatial derivatives are used to calculate kinetic energy -->
    <name>derivs</name>
    <type>complex</type>
    <components>deriv1 deriv2</components>
    <fourier_space>no</fourier_space>
    <![CDATA[
        deriv1 = 0;
        deriv2 = 0;
    ]]>
  </vector>

  <vector>
    <name>coefficients</name>
    <type>complex</type>
    <components>transmission reflection</components>
    <fourier_space>no</fourier_space>
    <![CDATA[
        transmission = 0;
        reflection = 0;
      ]]>
    </vector>

  </field>

  <!-- The sequence of integrations to perform -->
  <sequence>
    <!-- setting the incident field moving righward -->
    <filter>
      <vectors>main</vectors>
      <fourier_space>no</fourier_space>
      <![CDATA[
          psi1 *= rcomplex(cos(k*x),sin(k*x));
      ]]>
    </filter>

    <sequence>
      <cycles>400</cycles>
      <!-- each cycle is 0.4ns. 200 timesteps taken each cycle -->
      <!-- sampling at end of each cycle -->
      <integrate>
        <algorithm>RK4IP</algorithm>
        <lattice>199</lattice>
        <samples>0 0 0 0</samples>
        <interval>0.000398</interval>
        <k_operators>
          <constant>yes</constant>
          <operator_names>L1 L2</operator_names>
          <![CDATA[
              L1 = rcomplex(0,-kx*kx*hbar/(2*m1));
              L2 = rcomplex(0,-kx*kx*hbar/(2*m2));
          ]]>
        </k_operators>
        <vectors>main potential</vectors>
        <![CDATA[
            dpsi1_dt = L1[psi1] - 1/hbar *
                       rcomplex(0,     g1*mod2(psi1) + gc1*mod2(psi2))*psi1;
            dpsi2_dt = L2[psi2] - 1/hbar *
                       rcomplex(0, V + g2*mod2(psi2) + gc2*mod2(psi1))*psi2;
        ]]>
      </integrate>

      <filter>
        <!-- evaluating spatial derivatives, used to calculate kinetic energy -->
        <vectors>main derivs</vectors>
        <fourier_space>yes</fourier_space>
        <![CDATA[
            deriv1 = rcomplex(0,kx)*psi1;
            deriv2 = rcomplex(0,kx)*psi2;
        ]]>
      </filter>

      <filter>
        <vectors>main coefficients</vectors>
        <fourier_space>yes</fourier_space>
        <![CDATA[
            //These are just step functions to split the
            //field into left moving and right moving waves.
            //The "+1e-10" is to prevent division by zero and
            //count the kx = 0 wave as right moving.
            transmission = (kx/fabs(kx+1e-10)+1)/2*psi1;
            reflection = (-kx/fabs(kx+1e-10)+1)/2*psi1;
        ]]>
      </filter>

      <!-- This step samples everything once every 200 timesteps -->
      <integrate>
        <algorithm>RK4IP</algorithm>
      <lattice>1</lattice>
      <samples>1 1 1 1</samples>
      <interval>0.000002</interval>
      <k_operators>
        <constant>yes</constant>
        <operator_names>L1 L2</operator_names>
        <![CDATA[
            L1 = rcomplex(0,-kx*kx*hbar/(2*m1));
            L2 = rcomplex(0,-kx*kx*hbar/(2*m2));
        ]]>
      </k_operators>
      <vectors>main potential derivs coefficients</vectors>
      <![CDATA[
          dpsi1_dt = L1[psi1] - 1/hbar *
                     rcomplex(0,     g1*mod2(psi1) + gc1*mod2(psi2))*psi1;
          dpsi2_dt = L2[psi2] - 1/hbar *
                     rcomplex(0, V + g2*mod2(psi2) + gc2*mod2(psi1))*psi2;
      ]]>
    </integrate>

  </sequence>
</sequence>

<output format = "ascii" precision = "double">

  <!-- output the particle density -->
  <group>
    <sampling>
          <fourier_space> no </fourier_space>
          <lattice> 2048 </lattice>
          <moments>density1 density2</moments>
          <type>double</type>
          <![CDATA[
              density1 = mod2(psi1);
              density2 = mod2(psi2);
          ]]>
    </sampling>
  </group>

  <!-- output the energy per atom, in all its forms -->
  <group>
    <sampling>
      <vectors>main potential derivs</vectors>
          <fourier_space> no </fourier_space>
          <lattice>0</lattice>
          <moments>kinetic1 kinetic2 potential2 self1 self2 coup1 coup2 tot1 tot2</moments>
          <type>double</type>
          <![CDATA[
              //"lattice = 0" means integrate over x,
              //this gives the total energy (per atom).
              kinetic1 = 1/N1_1D*hbar*hbar/(2*m1)*mod2(deriv1);
              kinetic2 = 1/N2_1D*hbar*hbar/(2*m2)*mod2(deriv2);
              potential2 = 1/N2_1D*V*mod2(psi2);
              self1 = 1/N1_1D*g1/2*mod2(psi1)*mod2(psi1);
              self2 = 1/N2_1D*g2/2*mod2(psi2)*mod2(psi2);
              coup1 = 1/N1_1D*gc1/2*mod2(psi1)*mod2(psi2);
              coup2 = 1/N2_1D*gc2/2*mod2(psi1)*mod2(psi2);
              tot1 = kinetic1 +              self1 + coup1;
              tot2 = kinetic2 + potential2 + self2 + coup2;
          ]]>
    </sampling>
  </group>

  <!-- output the fourier transforms of the fields -->
  <group>
    <sampling>
          <fourier_space> yes </fourier_space>
          <lattice> 4096 </lattice>
          <moments>fourier1 fourier2</moments>
          <type>double</type>
```

```
            <![CDATA[
                fourier1 = psi1;
                fourier2 = psi2;
            ]]>
      </sampling>
    </group>

    <!-- output the reflected and transmitted proportions -->
    <group>
      <sampling>
      <vectors>coefficients</vectors>
            <fourier_space> no </fourier_space>
            <lattice> 0 </lattice>
            <moments>refl_coeff trans_coeff</moments>
            <type>double</type>
            <![CDATA[
                //"lattice = 0" means integrate over x - this gives the
                //total reflected and transmitted fractions
                refl_coeff =  mod2(reflection)/N1_1D;
                trans_coeff = mod2(transmission)/N1_1D;
            ]]>
      </sampling>
    </group>

  </output>
</simulation>
```

run.py

```
#! /usr/bin/env python
from pylab import *
import os
from matplotlib.ticker import MaxNLocator, NullLocator
import time

k_B = 1.38e-23
tmesh = 400
xmesh = 2048
kmesh = 4096

def BEC_collision(k):

    print "running 'coupled' ...",
    if os.system("./coupled -k " + str(k)) == 0:
        print "done."
    else:
        print "error running 'coupled'!"
        return

    print "generating datafiles from xsil file...",
    os.system('xsil2graphics coupled.xsil')
    print "generated."
    print "loading first datafile (density data)...",
    dat = loadtxt('coupled1.dat', delimiter=' ')
    print "loaded."
    t = dat[:,0]
    x = dat[:,1]
    n1 = dat[:,2]
    n2 = dat[:,3]
    del dat
    t = t.reshape([tmesh,xmesh])
    x = x.reshape([tmesh,xmesh])
```

```
n1 = n1.reshape([tmesh,xmesh])
n2 = n2.reshape([tmesh,xmesh])

print "loading second datafile (energy data)...",
dat = loadtxt('coupled2.dat', delimiter=' ')
print "loaded."
kin1  = dat[:,1]*1e9/k_B  #all converted to nanokelvin
kin2  = dat[:,2]*1e9/k_B
pot2  = dat[:,3]*1e9/k_B
self1 = dat[:,4]*1e9/k_B
self2 = dat[:,5]*1e9/k_B
coup1 = dat[:,6]*1e9/k_B
coup2 = dat[:,7]*1e9/k_B
tot1  = dat[:,8]*1e9/k_B
tot2  = dat[:,9]*1e9/k_B
del dat

print "loading third datafile (Fourier transforms) ...",
dat = loadtxt('coupled3.dat', delimiter = ' ')
print "loaded."
k_x = dat[:,1]
f_psi1 = dat[:,2]
f_psi2 = dat[:,3]
del dat
k_x = k_x.reshape([tmesh,kmesh])
f_psi1 = f_psi1.reshape([tmesh,kmesh])
f_psi2 = f_psi2.reshape([tmesh,kmesh])

print "loading fourth datafile (transmission and reflection coefficients) ...",
dat = loadtxt('coupled4.dat', delimiter = ' ')
print "loaded."
refl = dat[:,1]*100    #both as a percentage
trans  = dat[:,2]*100
del dat

#rescale time to milliseconds:
t = t[:,0]*1e3
#rescale wavenumbers to millions of rad per metre
k_x = k_x[0,:]/1e6

#plotting the results!
fig = figure(figsize = [12.8,8])

#the main plot:
axes([0.05, 0.05, 0.9, 0.325])
xlabel("position ($\mu$m)")
ylabel("particle density (x10$^{12}$ cm$^{-3}$)")
title("colliding BECs")
d1, = plot(x[0,:]*1e6,n1[0,:]/1e18)
d2, = plot(x[0,:]*1e6,n2[0,:]/1e18)
axis([x[0,1]*1e6,x[0,-2]*1e6,0,120])
grid(True)

#the energy plots for the incident BEC:
#coupling:
ax1 = axes([0.05, 0.45, 0.4, 0.125])
xlabel("time (ms)")
ylabel("coupling")
c1, = plot(t[:0],coup1[:0])
axis([t[1],t[-2],min(coup1),max(coup1)])
ax1.yaxis.set_major_locator(MaxNLocator(5))
grid(True)

#self:
ax2 = axes([0.05, 0.575, 0.4, 0.125])
ylabel("self")
s1, = plot(t[:0],self1[:0])
axis([t[1],t[-2],min(self1),max(self1)])
ax2.yaxis.set_major_locator(MaxNLocator(5))
grid(True)
```

```
#kinetic:
ax3 = axes([0.05, 0.70, 0.4, 0.125])
ylabel("kinetic")
k1, = plot(t[:0],kin1[:0])
axis([t[1],t[-2],min(kin1),max(kin1)])
ax3.yaxis.set_major_locator(MaxNLocator(5))
title("Incident BEC energy (nK)")
grid(True)


#the energy plots for the target BEC:
#coupling:
ax4 = axes([0.55, 0.45, 0.4, 0.125])
xlabel("time (ms)")
ylabel("coupling")
c2, = plot(t[:0],coup2[:0],color = 'green')
axis([t[1],t[-2],min(coup2),max(coup2)])
ax4.yaxis.set_major_locator(MaxNLocator(5))
grid(True)

#self:
ax5 = axes([0.55, 0.575, 0.4, 0.125])
ylabel("self")
s2, = plot(t[:0],self2[:0],color = 'green')
axis([t[1],t[-2],min(self2),max(self2)])
ax5.yaxis.set_major_locator(MaxNLocator(5))
grid(True)

#potential:
ax6 = axes([0.55, 0.70, 0.4, 0.125])
ylabel("potential")
p2, = plot(t[:0],pot2[:0],color = 'green')
axis([t[1],t[-2],min(pot2),max(pot2)])
ax6.yaxis.set_major_locator(MaxNLocator(5))
grid(True)

#kinetic:
ax7 = axes([0.55, 0.825, 0.4, 0.125])
ylabel("kinetic")
k2, = plot(t[:0],kin2[:0],color = 'green')
axis([t[1],t[-2],min(kin2),max(kin2)])
ax7.yaxis.set_major_locator(MaxNLocator(5))
title("Target BEC energy (nK)")
grid(True)

#putting the time and reflection coeff on the plot:
timecounter = figtext(0.40,0.3,"t = " + str(round(t[0],1)) + "ms")
reflection  = figtext(0.52,0.3,"R = " + str(round(refl[0],2)) + "%")

#the Fourier transform plot:
ax8 = axes([0.05,0.225,0.3,0.15])
xlabel('rad per metre (millions)')
f1, = plot(k_x,f_psi1[0,:])
axis([min(k_x),max(k_x),min(f_psi1.flatten()),max(f_psi1.flatten())])
ax8.yaxis.set_major_locator(NullLocator())
grid(True)

#the reflection/transmission plot:
ax9 = axes([0.65,0.225,0.3,0.15])
xlabel('time (ms)')
t1, = plot(t[:0],trans[:0],color = "blue")
r1, = plot(t[:0],refl[:0], color = "red")
axis([t[1],t[-2],-5,110])
grid(True)

#iterating over all time to make a series of plots:
print "generating", len(t), "frames, up to frame ...",
for i in range(len(t)):
    if i % 10 == 0:
            print i,
        d1.set_ydata(n1[i,:]/1e18)
        d2.set_ydata(n2[i,:]/1e18)
        k1.set_xdata(t[:i])
        k1.set_ydata(kin1[:i])
        k2.set_xdata(t[:i])
        k2.set_ydata(kin2[:i])
        p2.set_xdata(t[:i])
        p2.set_ydata(pot2[:i])
        s1.set_xdata(t[:i])
        s1.set_ydata(self1[:i])
        s2.set_xdata(t[:i])
        s2.set_ydata(self2[:i])
        c1.set_xdata(t[:i])
        c1.set_ydata(coup1[:i])
        c2.set_xdata(t[:i])
        c2.set_ydata(coup2[:i])
        f1.set_ydata(f_psi1[i,:])
        t1.set_xdata(t[:i])
        t1.set_ydata(trans[:i])
        r1.set_xdata(t[:i])
        r1.set_ydata(refl[:i])

        #display the time on the figure:
        fig.texts.remove(timecounter)
        timecounter = figtext(0.40,0.3,"t = " + str(round(t[i],1)) + "ms")

        #display the relection coefficient on the figure:
        fig.texts.remove(reflection)
        reflection  = figtext(0.52,0.3,"R = " + str(round(refl[i],2)) + "%")

        #prefixing zeros to the filename for correct ordering:
        number = str(i)
        while len(number) < 4:
            number = "0" + number

        #outputting a figure:
        savefig('images/coupled' + number + '.png')

    print "\n", len(t), "frames generated."
    #making a movie:
    print "making a movie ...",
    os.system("mencoder mf://images/*.png -mf " +
                  "type=png:w=1280:h=800:fps=20 " +
                  "-ovc lavc -lavcopts vcodec=mpeg4 " +
                  "-oac copy -o videos/" + str(k) + "_rad_per_metre.avi")
    print str(k) + "_rad_per_metre.avi saved."
    print "done!"



k_array = linspace(3.05,6.95,100)*1e6
print "simulation started at:", time.strftime('%X')

print "compiling code...",
if os.system("xmds initial.xmds") == 0:
    print "'inital' compiled ok ...",
else:
    print "error compiling initial.cc!"

if os.system("xmds coupled.xmds") == 0:
    print "'coupled' compiled ok."
else:
    print "error compiling 'coupled'!"

print "initializing BECs ...",
if os.system("./initial") == 0:
    print "initialized."
else:
    print "error initializing!"
for i in range(len(k_array)):
    k = k_array[i]
    print "\n\nsmashing BECs together with k = ", k, "rad per metre."
    print "simulation started at:", time.strftime('%X')
    BEC_collision(k)

print "finishing time:", time.strftime('%X')
print"Total of", len(k_array), "simulations performed."
```