Vortex Interactions in Type II Superconductors

Brian Martin

April 26, 2005

University of British Columbia
Physics 449: Honours Thesis

Abstract

In describing the properties of type II superconductors there is an important parameter a, called the scattering length. It is related to another parameter g, which determines certain properties of the vortex system (see ref.[1] for details). The goal of this project was to find the value of a. The first approximation to the solution found a to be ≈ 15.5. Numerical difficulties and time constraints yielded inaccurate results for the second approximation to the solution.

Contents

1 Introduction

In this section the important background information is discussed. The relevant topics are superconductors (specifically, type II superconductors) and a brief discussion of vortex interactions. References for a more detailed development of the ideas are given in the appropriate sections. The following sections are based heavily on De Gennes' work [2].

1.1 Superconductors

Below certain low temperatures ($\approx 10K$), some metals enter a new thermodynamic state. X-ray crystallography confirms that it is not a new crystal structure, and neutron scattering confirms that it is not a magnetic transition. The change is that at low temperatures these metals show no electrical resistance. This state is called the superconducting state. For example, a current induced in a ring of tin at $T < 3.7K$ has been observed to persist for over a year. Mercury was the first superconductor to be discovered by Kammerling Onnes in 1911.

The free energy curve for a superconductor can be derived from specific heat data. The data curve for the superconducting state meets the curve for the normal state at the transition temperature (T_0). At $T = 0$, the difference between the normal free energy (F_N) curve and the superconducting (F_S) curve is of the order $(k_BT_0)^2/E_f$, where E_f is the Fermi Energy of the conduction electrons in the normal metal. (Some typical values: $E_f \approx 1eV$, and $k_BT_0 \approx 10^{-3}eV$.)

We now consider what effects super-currents ($\mathbf{j}(\mathbf{r})$) and their corresponding magnetic fields ($\mathbf{h}(\mathbf{r})$) have on the free energy. The equation for the free energy is:

$$F = \int F_s d\mathbf{r} + E_{kin} + E_{mag} \tag{1}$$

F_s is the energy of the electrons in the superconducting state, E_{kin} is the energy associated with the currents, and E_{mag} comes from the magnetic field.

Let $\mathbf{v}(\mathbf{r})$ be the drift velocity of the electrons and n_s be the density of superconducting electrons. Then the current is related to the drift velocity and density by:

$$n_s e\mathbf{v}(\mathbf{r}) = \mathbf{j}_s(\mathbf{r}) \tag{2}$$

where e is the charge of the electron. The kinetic energy in eq. 1 is given by

$$E_{kin} = \int d\mathbf{r}\frac{1}{2}m\mathbf{v}(\mathbf{r})^2 n_s \tag{3}$$

where the integral is taken over the volume of the superconductor. Equation 3 is approximately correct for slowly varying $\mathbf{v}$, and is exact for $\mathbf{v} = \textit{constant}$. The energy associated with the magnetic field is given by

$$E_{mag} = \int \frac{\mathbf{h}(\mathbf{r})^2}{8\pi} d\mathbf{r} \tag{4}$$

We also have Maxwell's equation which relates the magnetic field and the current:

$$\nabla \times \mathbf{h} = \frac{4\pi}{c}\mathbf{j}_s \tag{5}$$

Now we can rewrite eq. 1 the following way:

$$E = \int F_s d\mathbf{r} + \frac{1}{8\pi}\int d\mathbf{r}[\mathbf{h}^2 + \lambda_L^2|\nabla \times \mathbf{h}|^2] \tag{6}$$

$$\lambda_L = \left[\frac{mc^2}{4\pi n_s c^2}\right]^{\frac{1}{2}} \tag{7}$$

We seek to find the distribution of the magnetic field that will minimize the free energy. The change in the free energy corresponding to a change in the magnetic field is:

$$\delta E = \frac{1}{4\pi}\int[\mathbf{h}\cdot\delta\mathbf{h} + \lambda_L^2(\nabla\times\mathbf{h})\cdot(\nabla\times\delta\mathbf{h})]d\mathbf{r} \tag{8}$$

$$= \frac{1}{4\pi}\int[\mathbf{h} + \lambda_L^2\nabla\times\nabla\times\mathbf{h}]\cdot\delta\mathbf{h}d\mathbf{r} \tag{9}$$

The condition that minimizes E is when $\delta E = 0$. The occurs when the integrand is 0; therefore we arrive at the London Equation:

$$\mathbf{h} + \lambda_L^2\nabla\times\nabla\times\mathbf{h} = 0 \tag{10}$$

The London equation combined with Maxwell's equations (eq. 5) and

$$\nabla\cdot\mathbf{h} = 0 \tag{11}$$

allow us to determine the field and current distribution inside the superconductor.

As a simple example, let us apply the London equation to a semi-infinite slab of superconducting material. Let the face of the slab be the xy plane, such that $z < 0$ is the superconductor, and $z > 0$ is a vacuum. Assume $\mathbf{h}$ and $\mathbf{j}$ depend only on z. If $\mathbf{h}$ is parallel to the z axis, then eq. 11 becomes $\partial h/\partial z = 0$, and h is constant. Eq. 5 then gives $\nabla\times\mathbf{h} = 0$, and therefore $\mathbf{j}_s = 0$, so $\mathbf{h} = 0$. Therefore, a magnetic field parallel to z (normal to the surface) is not allowed.

If $\mathbf{h}$ is perpendicular to z (tangential to the surface), then eq. 11 is automatically satisfied. If the axes are then chosen such that $\mathbf{h}$ is parallel to the x axis, then $\mathbf{j}_s$ must be parallel to the y axis and is given by:

$$\frac{dh}{dz} = \frac{4\pi j_s}{c} \tag{12}$$

Putting this into eq. 10 gives:

$$\frac{d^2h}{dz^2} = \frac{h}{\lambda_L^2} \tag{13}$$

The solutions to this equation are exponentials, but the only finite solution is exponentially decreasing:

$$h(z) = h(0)e^{-z/\lambda_L} \tag{14}$$

This is why λ_L is called the penetration depth, because the field only penetrates the superconductor that far. The most important conclusion from this example is that it is energetically favorable to completely expel magnetic flux lines. This was experimentally shown by Meissner and Ochsenfeld in 1933, and it is called the Meissner effect. However, these results are valid only for weak applied magnetic fields and for macroscopic samples.

The derivation for the London equation assumed a weak magnetic field and a slowly varying drift velocity function. We require that $v(r)$ be correlated between two neighboring electrons. This will be the case if $v(r)$ has negligible variation over the distance between two electrons. Another way of saying this is that the distance between neighboring electrons is less than the correlation length, ξ_0. The correlation length is defined by

$$\xi_0 = \frac{\hbar v_f}{\pi \Delta} \tag{15}$$

where v_f is the velocity of electrons at the Fermi energy level, and Δ is the excitation energy. It is the energy required to take an electron from below the Fermi energy and excite it. For non-superconducting metals, Δ is quite small, but it gets much larger for metals in the superconducting state.

For stronger applied magnetic fields, superconductors can be split into two classes. Type I superconductors are characterized by smaller penetration depths ($\lambda_L \approx 300\text{Å}$) and large v_f ($> 10^8 cm/s$). As can be seen from eq. 15, this gives a large correlation length ($\approx 10^4\text{Å}$). Therefore, the London equation does not apply to type I superconductors, as $\mathbf{v}$ is not constant over such a large length. These are usually simple, non-transition metals. These materials still exhibit the Meissner effect, but it is as a result of a somewhat more complicated equation.

Type II supercondcutors are at the opposite end of the spectrum. They are usually transition metals and compounds like Nb_3Sn and V_3Ga. These metals have larger penetration depths ($\lambda_L \approx 2000$Å) and smaller velocities ($v_f \approx 10^6 cm/s$), which gives a much smaller coherence length, $\xi \approx 50$Å. Therefore the London equation is valid in type II superconductors. These are the type to be studied in this paper.

1.2 Type II Superconductors

The previous discussion only dealt with weak applied magnetic fields. Type I and II superconductors also differ in their response to slightly stronger fields. For type I superconductors, there is an abrupt change between the complete Meissner effect phase and the normal phase. However, type II superconductors exhibit an intermediate state.

De Gennes shows that the critical field strength at which the transition between the superconducting and normal state occurs is determined by $F_n - F_s = H_c^2/8\pi$, where F_n is what the free energy would be if the material was normal, and F_s is the actual value of the free energy calculated from the specific heat data. Experimentally it is found that type II superconductors only exhibit complete Meissner effect below a much weaker magnetic field, H_{c1}. For $H > H_{c1}$, the magnetic flux lines penetrate, but not completely. The flux through the slab is less than it would be in the normal state (that is, at a higher temperature). This state exists for $H_{c1} < H < H_{c2}$. For $H > H_{c2}$, the material does not show any flux expulsion on the macroscopic scale, but there is still superconductivity on the surface (this can be measured by placing nodes on the surface and measuring the resistance).

The existence of this partial penetration region was first shown by Schubnikov in 1937. There are regions that are normal inside regions that are superconduting. There are two possible situations for the arrangement of these normal regions: either they will be lamina of thickness $\approx \xi_0$, or they will be filaments ("cylinders") of radius $\approx \xi_0$. Theoretical calculations show that the latter situation is preferred.

Each filament has a core of radius $\approx \xi_0$. The magnetic field is a maximum at the center of the filament but extends a distance $\approx \lambda_L$. Circular currents surround the filament (due to the magnetic field) and screen out the field for distances $r \approx \lambda_L$.

We now wish to compute the line energy along a vortex line (ignoring the complicated structure occurring when $r < \xi_0$). The equation for the energy is:

$$\tau = \int_{r>\xi_0} dr \frac{1}{8\pi}[h^2 + \lambda^2 |\nabla \times h|^2] \tag{16}$$

De Gennes shows that the form of h that minimizes the energy is

$$h = \frac{\phi_O}{2\pi\lambda^2} K_0 \left(\frac{r}{\lambda}\right) \tag{17}$$

where $K_0(x)$ is the zeroth order modified Bessel function. Then the form of the free energy equation (eq. 16) is

$$\tau = \left(\frac{\phi_0}{2\pi\lambda}\right)^2 ln\left(\frac{\lambda}{\xi}\right) \tag{18}$$

where ϕ_0 is the flux quantum, λ is the penetration depth, and ξ is the coherence length. The most important result of this equation is that τ is a quadratic function of the flux quantum ϕ_0. This means that it is more favorable to have two flux lines (energy 2τ) instead of a single flux line with twice the flux (energy 4τ).

Now let us modify the London equation (eq. 10) to account for the hard core of the vortex at $r < \xi$. De Gennes suggests we modify the equation by adding a 2-d delta function:

$$h + \lambda^2 \nabla \times \nabla \times h = \phi_0 \delta^2(r) \tag{19}$$

The constant in front of the delta function shows that the vortex carries a flux of ϕ_0.

1.3 Interaction of Vortex Lines

The next important topic is the interaction of vortices with each other. Suppose we have two parallel vortices lined up along the z axis at positions $r_1 = (x_1, y_1)$ and $r_2 = (x_2, y_2)$. Then eq. 19 becomes

$$h + \lambda^2 \nabla \times \nabla \times h = \phi_0 [\delta(\mathbf{r} - \mathbf{r_1}) + \delta(\mathbf{r} - \mathbf{r_2})] \tag{20}$$

where $h = h_1 + h_2$ is now the sum of two of the fields given by eq. 17. The free energy has the same form as eq. 16 but now we integrate over the surface of the two vortices, $d\sigma_1$ and $d\sigma_2$:

$$\begin{aligned} F = & \int dr \frac{1}{8\pi}[h^2 + \lambda^2 |\nabla \times h|^2] \\ = & \frac{\lambda^2}{8\pi} \int h \times \nabla \times h \cdot d\sigma \end{aligned} \tag{21}$$

The integral over $d\sigma$ is for $|\mathbf{r} - \mathbf{r_i}| < \xi$. If we explicitly show the two domains of integration we get

$$F = \frac{\lambda}{8\pi} \int (d\sigma_1 + d\sigma_2) \cdot (h_1 + h_2) \times (\nabla \times h_1 + \nabla \times h_2) \tag{22}$$

There are a total of eight terms here. We will group them together in the following way: the individual line energies are

$$2\tau = \frac{\lambda^2}{8\pi} \left[\int d\sigma_1 \cdot h_1 \times \nabla \times h_1 + \int d\sigma_2 \cdot h_2 \times \nabla \times h_2 \right]; \tag{23}$$

there are four terms that go to zero in the limit that $\xi \to 0$:

$$\int (h_1 + h_2) \cdot (\nabla \times h_1 \times d\sigma_2 + \nabla \times h_2 \times d\sigma_1)) \tag{24}$$

because $\nabla \times h_1$ is finite in the $d\sigma_2$ region (and vice versa), so as $d\sigma_2 \to 0$ (because $\xi \to 0$), the entire integral vanishes. The last group of terms is important, it represents the interaction energy:

$$U_{12} = \frac{\lambda^2}{8\pi} \int (h_1 \times \nabla \times h_2 \cdot d\sigma_2 + h_2 \times \nabla \times h_1 \cdot d\sigma_1) \tag{25}$$

If we define h_{12} by

$$h_{12} = h_1(r_2) = h_2(r_1) = \frac{\phi_0}{2\pi\lambda^2} K_0 \left(\frac{r_1 - r_2}{\lambda} \right) \tag{26}$$

then we can set

$$U_{12} = \frac{\phi_0 h_{12}}{4\pi} \tag{27}$$

Finally, the expression for the free energy becomes:

$$F = 2\tau + \frac{\phi_0 h_{12}}{4\pi} \tag{28}$$

We ignore the first term, because it is independent of the distance, and the free energy F we use in the path integral approach is

$$F = \frac{\phi_0^2}{8\pi^2\lambda^2} K_0 \left(\frac{r_1 - r_2}{\lambda} \right) \tag{29}$$

1.4 Problem Statement

The problem to be solved involves the interaction of two vortices in a slab (see fig. 1). As will be shown later in the path integrals section, this interaction can be solved like a quantum mechanical Schrodinger equation problem.

There are three distances in this problem. The x distance is the horizontal separation of the two vortices. The y distance is how far off the middle of the slab vortex 1 is. The z distance is the same but for particle 2. The boundary conditions are that the particle cannot exist on the y (or z for the other

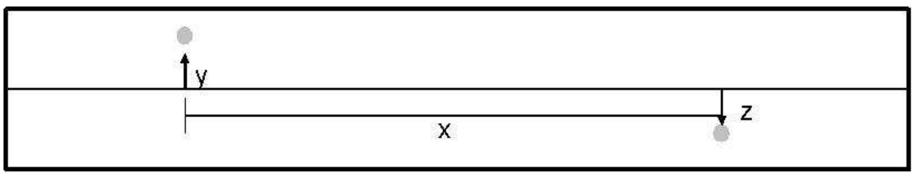


Figure 1: This picture is of the problem domain. The x distance represents the horizontal separation of the two vortices. The y and z distances are the distances from the horizontal axis.

particle) boundary. To enforce this condition, image vortices are introduced, as is standard procedure in similar electricity and magnetism problems [6]. Because there are both upper and lower boundaries, an infinite number of image charges are required. Fortunately, only a small number of them contribute significantly.

The potential is a sum of the interactions of each vortex with all the images. Each particle interacts with its own image charges (fig. 2) and with the other particle's images (fig. 3). It is shown later (eq. 69) that the effective potential has a multiplicative factor in front of it. The solution was found for various values of this constant according to the equation:

$$C = 0.27 * 10^{\alpha} \tag{30}$$

Dr. Affleck proposed that the scattering length should be logarithmically proportional to C, which is the same as being linearly proportional to α. The actual value of C is $0.27 * 10^6$, which is found from the actual constants in eq. 29.

The first method of solving involves the approximation that the vortices stay in the middle of the slab. The eigenvalue and initial value problem method are both used to handle this case. Then that assumption is relaxed and the new problem is a function of three variables (x, y, and z) which is handled by Gauss-Seidel relaxation.

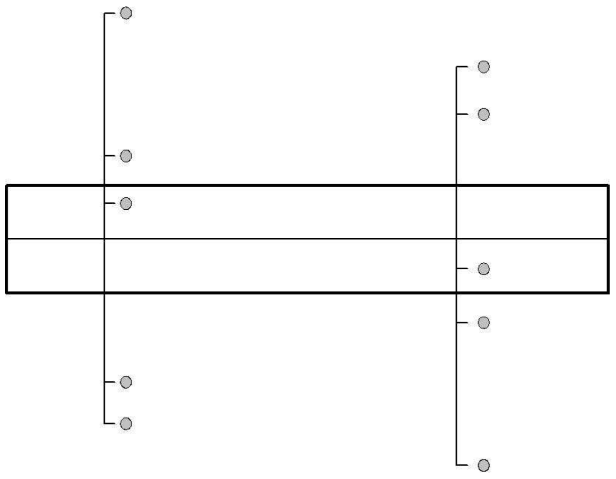


Figure 2: This figure shows the distances from the vortices (inside the slab) to their images (outside the slab). Note that there are actually an infinite number of images but only the first few are shown.

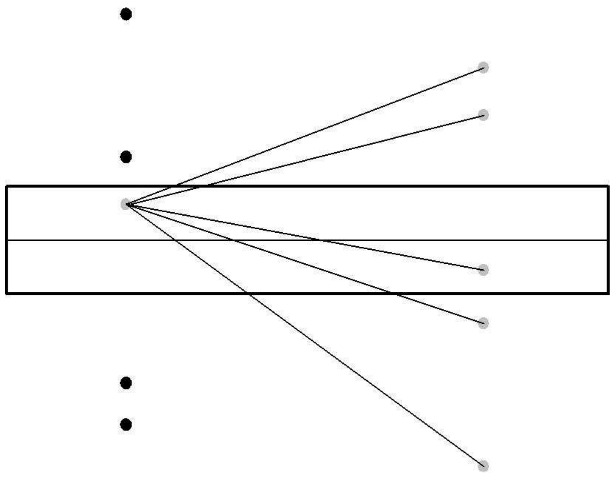


Figure 3: This figure shows the distances from the vortices (inside the slab) to the other vortex's images (outside the slab). Note that there are actually an infinite number of images but only the first few are shown.

2 Theory

This section discusses the mathematical and numerical concepts needed to understand the rest of the paper. Feynman's theory of path integration is reviewed first, and then a description of the numerical methods being used is given.

2.1 Path Integrals

The following sections closely follow Feynman's description (see ref [3]).

2.1.1 The Classical Action

Classical mechanics is concerned with describing how objects can travel from one place to another. There are infinitely many ways to get from point A to point B, but the actual path is determined by the Principle of Least Action. Let us define S as the action. The Principle of Least Action says that the path chosen is the one that extremizes S, that is $S' = 0$. Let the path be $\vec{x}(t)$, and the object is at $\vec{x}_a$ at $t_a$and at $\vec{x}_b$ at t_b. Then S is defined as:

$$S = \int_{t_a}^{t_b} L(\dot{x}, x, t)dt \tag{31}$$

where L is the Lagrangian of the system. For a particle of mass m and moving in a potential that is a function of both position and time $V(x,t)$, the Lagrangian is:

$$L = \frac{m}{2}\dot{x}^2 - V(x,t) \tag{32}$$

We require that the variation in S be 0 to first order in $\delta\vec{x}$ to determine the path taken. The variation in S is given by:

$$\delta S = S[\vec{x} + \delta\vec{x}] - S(\vec{x}) \tag{33}$$

and:

$$S[\vec{x}+\delta\vec{x}] = \int_{t_a}^{t_b} L(\dot{x}+\delta\dot{x}, x+\delta x, t)dt - \int_{t_a}^{t_b} L(\dot{x}, x, t)dt \quad (34)$$

The first order expansion of the first integral gives:

$$\begin{aligned} \delta S = & \int_{t_a}^{t_b} [L(\dot{x}, x, t) + \delta\dot{x}\frac{\partial L}{\partial \dot{x}} + \delta x \frac{\partial L}{\partial x}]dt - \int_{t_a}^{t_b} L(\dot{x}, x, t)dt \quad (35) \\ = & \int_{t_a}^{t_b} [\delta\dot{x}\frac{\partial L}{\partial \dot{x}} + \delta x \frac{\partial L}{\partial x}]dt \quad (36) \end{aligned}$$

Upon integration by parts, this gives:

$$\delta S = \delta x \frac{\partial L}{\partial \dot{x}}|_{t_a}^{t_b} - \int_{t_a}^{t_b} \delta x (\frac{d}{dt}\frac{\partial L}{\partial \dot{x}} - \frac{\partial L}{\partial x})dt \quad (37)$$

but we have kept the endpoints fixed, such that $\delta x = 0$ at t_a and t_b. Thus we end up with the Lagrangian equation of motion:

$$\frac{d}{dt}(\frac{\partial L}{\partial \dot{x}}) - \frac{\partial L}{\partial x} = 0 \quad (38)$$

2.1.2 Quantum Mechanics

The difference between the classical path previously mentioned and the quantum mechanical path is that classically only one path is allowed, while quantum mechanics relaxes that restriction. The Principle of Least Action determines the path between a and b classically. But in Quantum Mechanics, all paths contribute, and we must find out by how much.

The probability to go from x_a at t_a to x_b at t_b is the absolute square of the kernel $K(b, a)$:

$$P(b, a) = |K(b, a)|^2 \quad (39)$$

where $K(b, a)$ is defined by a sum over all paths from a to b.

$$K(b, a) = \Sigma_{allpaths}\phi[x(t)] \quad (40)$$

where ϕ is the amplitude of the specific path and is defined by:

$$\phi[x(t)] = Ce^{\frac{i}{\hbar}S[x(t)]} \tag{41}$$

where C is a constant chosen to normalize $K(b,a)$ appropriately.

This formula shows that each path contributes the same amount, but at different phases. This means the amplitudes of some paths will cancel when added, while others will increase the total amplitude.

2.1.3 The Sum Over All Paths

Feynman draws the analogy between the sum of all paths and the Reimann Integral. The area under a curve is proportional to the sum of all the values of the function. To make this statement more clear, we choose a subset of all the values, say those separated by a distance h. We then say that the area is proportional to that sum:

$$A \sim \Sigma_i f(x_i) \tag{42}$$

Next, we take more and more points to approximate A better. We must introduce a normalizing factor, otherwise the sum will grow without bound. So we say:

$$A = h\Sigma_i f(x_i) \tag{43}$$

and we take the limit as $h \to 0$:

$$A = lim_{h\to 0} h\Sigma_i f(x_i) \tag{44}$$

We use a similar procedure to define the sum over all paths. First, choose a subset of time as it is the independent variable; take N values at width ϵ apart. At each time t_i, choose a position x_i, and then connect all neighboring points with straight lines. For a specific path,

$$K(b,a) \sim \phi[x(t)] \tag{45}$$

along that path. Then we integrate over all variations in x_1 to x_{N-1} leaving x_0 and x_N fixed, as they are the endpoints:

$$K(b,a) \sim \int\int \cdots \int \phi[x(t)]dx_1 dx_2 \cdots dx_{N-1} \tag{46}$$

Then we take the separation ϵ smaller and smaller, but we again need a normalizing factor which will depend on ϵ, call it $A(\epsilon)$. Then we have:

$$K(b,a) = lim_{\epsilon \to 0} \frac{1}{A} \int\int \cdots \int e^{\frac{i}{\hbar}S[b,a]} \frac{dx_1}{A}\frac{dx_2}{A} \cdots \frac{dx_{N-1}}{A} \tag{47}$$

where

$$S[b,a] = \int_{t_a}^{t_b} L(\dot{x}, x, t)dt \tag{48}$$

is the line integral over the current path.

2.1.4 Path Integrals in Statistical Mechanics

The goal of this section is to show how the concepts of statistical mechanics can be formulated in terms of path integrals, and then to show that vortex interactions are a statistical mechanical problem, which can be represented by path integrals.

In statistical mechanics, the partition function plays a critical role in determining that properties of the system. For example, the probability that a system is in a state of energy E_i (assuming non-degenerate energy levels) is:

$$p_i = \frac{1}{Z} e^{-E_i \beta} \tag{49}$$

where Z is the partition function and

$$\beta = \frac{1}{kT}, \tag{50}$$

k being Boltzmann's constant. The derivation of these results can be found in most thermal physics or statistical mechanics texts (see ref.[4], for example).

It can also be shown that if O is some observable quantity, the statistical average for the whole system is

$$\bar{O} = \Sigma p_i O_i = \frac{1}{Z} O_i e^{-E_i \beta} \tag{51}$$

Let a particular state be defined by ϕ_i. Then,

$$\bar{O} = \frac{1}{Z} \Sigma_i \int \phi_i^*(x) O \phi_i(x) e^{-\beta E_i} dx \tag{52}$$

Feynman shows that any observable quantity can be found if we know the function $\rho(x', x)$, defined by:

$$\rho(x', x) = \Sigma_i \phi_i(x') \phi_i^*(x) e^{-\beta E_i}, \tag{53}$$

We let O operate on ϕ_i only, not on ϕ_i^*. So we use $\rho(x', x)$ and let O act only on x', then set $x' = x$ in the expression $O\rho(x', x)$ and integrate over all x, and we our left with equation 52.

Also note that if $P(x)$ is the probability of finding the system in state x, defined by

$$P(x) = \frac{1}{Z} \Sigma_i \phi_i^*(x) \phi_i(x) e^{-\beta E_i}, \tag{54}$$

then,

$$P(x) = \frac{1}{Z} \rho(x, x). \tag{55}$$

$\rho(x', x)$ is called the density matrix, or probability density matrix, and many problems in statistical mechanics reduce to determining $\rho(x', x)$ via eq. 53.

The equation for the density matrix looks like the definition of the Kernel as defined before. To go from state 1 to state 2, we have:

$$K(x_2, t_2; x_1, t_1) = \Sigma_i \phi_i(x_2) \phi_i^*(x_1) e^{-\frac{i}{\hbar} E_i (t_2 - t_1)}. \tag{56}$$

In fact, the expression for the density matrix is the same as the Kernel corresponding to an imaginary negative time interval.

Feynman shows that for a Hamiltonian H given by

$$H = -\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x), \tag{57}$$

the Kernal over a short time interval $\epsilon = t_2 - t_1$ is,

$$K(2,1) = \sqrt{\frac{m}{2\pi i\hbar\epsilon}} exp[\frac{im}{2\hbar}\frac{(x_2 - x_1)^2}{\epsilon} - \frac{i}{\hbar}\epsilon V(\frac{x_2 + x_1}{2})], \tag{58}$$

and we produce a solution by developing a path integral by taking $\epsilon \to 0$ and a product of infinitely many Kernels.

So we have shown that you can map statistical mechanics problems into path integrals and shown a brief glimpse of how path integrals and the Schrodinger Equation are connected, but we need to develop that idea further. This will be done later on. For now, we turn to showing how vortex interactions can be represented as statistical mechanics problems.

2.1.5 Vortex Interactions and Statistical Mechanics

Nelson ([5]) has shown that in a system of N flux lines (vortices) whose positions are defined by $\vec{r_i}(z)$ $(i = 1...N)$, the Gibbs free energy is

$$G = (\epsilon_1 - \frac{H\Phi_0}{4\pi}) + \frac{\Phi_0^2}{8\pi^2\lambda^2}\Sigma_{i>j}\int_0^L K_0(\frac{\vec{r_{ij}}}{\lambda})dz + \frac{1}{2}\epsilon_1\Sigma_{i=1}^N\int_0^L |\frac{d\vec{r_i}(z)}{dz}|^2 dz, \tag{59}$$

where $\vec{r_{ij}} = \vec{r_i} = \vec{r_j}$, $K_0(x)$ is the modified Bessel Function, λ is the penetration depth, L is the length of the sample, and Φ_0 is the flux quantum:

$$\Phi_0 \frac{2\pi\hbar c}{2e} \tag{60}$$

Also, Affleck et al. ([1]) have shown that for a system of N flux lines, the partition function is:

$$Z = \frac{1}{N!}\Pi_{i=1}^N \int D[x_i(z)]e^{-F[x_i(z)]\beta} \tag{61}$$

where F is the free energy of the system. Therefore the problem of interacting vortices can be solved with statistical mechanics methods, which, in turn, can be solved by quantum mechanical methods.

2.1.6 From Path Integrals to the Schrodinger Equation

The path integrals that have been discussed so far have the property that the action S obeys:

$$S(2,1) = S(2,3) + S(3,1). \tag{62}$$

That is, the action from state 1 to state 2 is the same as the sum of the action from state 1 to state 3 and state 3 to state 2. Because this holds for any time interval, we can take the interval to be infinitesimal, and in this was formulate a differential equation. Equation 62 allows us to relate the value of a path integral to its value a short time later, which we then use to develop the differential equation. The wave function at any time can also be found from the wave function at a previous time via:

$$\psi(x_2, t_2) = \int_{-\infty}^{\infty} K(x_2, t_2; x_1, t_1)\psi(x_1, t_1)dx_1 \tag{63}$$

Feynman goes through the derivation of the differential equation for the case that the Lagrangian L is given by:

$$L = \frac{m\dot{x}^2}{2} - V(x,t), \tag{64}$$

and the resulting differential equation is the Schrodinger Equation:

$$-\frac{\hbar}{i}\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m}\frac{\partial^2\psi}{\partial x^2} + V(x,t)\psi \tag{65}$$

Now we see that vortex interaction problems are mathematically equivalent to solving the Schrodinger Equation.

2.2 Solving the Schrodinger Equation in One Dimension

In this section we review the details of the numerical methods used to solve the Schrodinger Equation in one dimension. The first method involves using an eigenvalue solver, as the Schrodinger equation is an eigenvalue problem. The second method involves making the approximation that the energy goes to zero, and in that case we can use an initial value method to find the solution.

All of the numerical methods discussed here have been learned from Dr. Matt Choptuik, either through classes or private conversations.

2.2.1 Eigenvalue Problem

The Schrodinger Equation is an eigenvalue problem; the Hamiltonian operator becomes a matrix when discretized, and we have

$$H\Psi = E\Psi \tag{66}$$

and we wish to solve for the eigenvalue E and the eigenvector Ψ. We discretize the continuum into a finite grid of points x_i on the problem domain $x_{min} < x < x_{max}$. If there are N grid points, then the distance between neighboring grid points is

$$h = \frac{x_{max} - x_{min}}{N - 1} \tag{67}$$

Let Ψ_i be the value of Ψ at x_i. Then, the second derivative at Ψ_i is given by:

$$\Psi_i'' = \frac{\Psi_{i-1} - 2\Psi_i + \Psi_{i+1}}{h^2} + O(h^2) \tag{68}$$

Ignoring the higher order terms, we are left with an approximation to the second derivative that is correct to second order.

Next, we group all the constants into one constant, C, and re-write the Schrodinger Equation as

$$-\Psi'' + CV\Psi = E\Psi \tag{69}$$

The parameter C is absorbed into V for now. Using the discretized version of the second derivative, eq. 68, we can re-write the Schrodinger equation at a single point x_i as

$$-\frac{\Psi_{i-1} - 2\Psi_i + \Psi_{i+1}}{h^2} + V\Psi_i = E\Psi_i \tag{70}$$

We then apply this equation at each point in the grid and enforce periodic boundary conditions by coupling the 1^{st} and N^{th} points. This produces the following matrix equation:

$$\begin{bmatrix} \frac{2}{h^2} + V_1 & -\frac{1}{h^2} & 0 & \cdots & 0 & -\frac{1}{h^2} \\ -\frac{1}{h^2} & \frac{2}{h^2} + V_2 & -\frac{1}{h^2} & 0 & \cdots & 0 \\ 0 & -\frac{1}{h^2} & \frac{2}{h^2} + V_3 & -\frac{1}{h^2} & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & 0 & \ddots & \ddots & -\frac{1}{h^2} \\ -\frac{1}{h^2} & 0 & \cdots & 0 & -\frac{1}{h^2} & \frac{2}{h^2} + V_N \end{bmatrix} [\Psi] = E\,[\Psi] \tag{71}$$

We can then put this matrix into an eigenvalue solver and it will give us the eigenvalues and associated eigenvectors. I used Fortran's lapack library.

The next step is to extract the scattering length from the solution. The solver gives us back an array of eigenvalues and another array of associated eigenvectors. We are interested in the ground state energy, so we find the lowest eigenvalue. The plots of the eigenvectors can be seen in the results section. The solution is expected to be of a sinusoidal form at large length scale:

$$\Psi = A\sin\left(k(|x| - a)\right) \tag{72}$$

The parameter we are interested in is the scattering length, represented by a in eq. 72. The solution can be fit to that form and then the scattering length can be extracted. This process will be described in the results section.

2.2.2 Initial Value Problem

Because we are interested in only the ground state wave function, the energy can be taken to go to zero. In this limit, the Schrodinger equation becomes an initial value problem instead of an eigenvalue problem:

$$-\Psi'' + CV\Psi = 0 \tag{73}$$

In the large x region, $V(x)$ drops off to zero and we are left with:

$$-\Psi'' = 0 \tag{74}$$

The solution of which is just a line. The line has the form:

$$\Psi = k(|x| - a) \tag{75}$$

Again, the scattering length is given by fitting the solution to this from and reading off the value of a.

The solving method used in this section requires that the equations come in first order canonical form. This means that instead of a single second order equation, we get two coupled first order equations:

$$\begin{aligned} y_1 = &\quad y_2' \\ y_1' = &\quad CVy_2 \end{aligned} \tag{76}$$

The solver then computes the solution at requested output times. Plots of the solution can be seen in the results section.

2.3 Solving the Schrodinger Equation in Three Dimensions

The next step is to relax the assumption that the particles stay in the middle of the slab. They can now each have a non-zero

y coordinate, which makes a total of three dimensions including the separation distance. This also means a new way of representing the image charges is needed in order to allow them to move in response to the movement of one of the particles. Now the potential is a function of three variables as well. The same second order approximation to the second derivative as before can be used but now it is generalized to three dimensions:

$$\begin{aligned}\Psi'' \approx \quad & \frac{\Psi_{i-1,j,k} - 2\Psi_{i,j,k} + \Psi_{i+1,j,k}}{h_x^2} \\ + \quad & \frac{\Psi_{i,j-1,k} - 2\Psi_{i,j,k} + \Psi_{i,j+1,k}}{h_y^2} \\ + \quad & \frac{\Psi_{i,j,k-1} - 2\Psi_{i,j,k} + \Psi_{i,j,k+1}}{h_z^2} \qquad (77)\end{aligned}$$

And the full Schrodinger equation reads:

$$\begin{aligned} & \frac{-\Psi_{i-1,j,k} + 2\Psi_{i,j,k} - \Psi_{i+1,j,k}}{h_x^2} \\ + \quad & \frac{-\Psi_{i,j-1,k} + 2\Psi_{i,j,k} - \Psi_{i,j+1,k}}{h_y^2} \\ + \quad & \frac{-\Psi_{i,j,k-1} + 2\Psi_{i,j,k} - \Psi_{i,j,k+1}}{h_z^2} + V_{i,j,k}\Psi_{i,j,k} = E\Psi_{i,j,k} \qquad (78)\end{aligned}$$

The method used to solve this problem is called Successive Over Relaxation. It is based on the Gauss-Seidel algorithm, which is defined next.

2.3.1 Gauss-Seidel Relaxation

The 3D problem cannot be easily put into a matrix form, nor can we use the initial value method anymore because the freedom in the y direction always contributes a non-zero energy. Instead we use the Gauss-Seidel algorithm. As before, the continuum domain is discretized, but this time in three dimensions. I use the convention that i represents an x index, j for y, and k for z. Then the domain is x_i, where $x_1 = x_{min}$ and $x_N = x_{max}$, And likewise for the other variables but with possibly different minimum and maximum values, as well as different number of grid points N. The algorithm works by looking at each lattice point individually. The discretized Schrodinger equation (eq. 77) can be solved exactly at that point according to:

$$\begin{aligned}\Psi_{i,j,k}\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}+\frac{2}{h_z^2}+V_{i,j,k}-E\right) = &\frac{\Psi_{i-1,j,k}+\Psi_{i+1,j,k}}{h_x^2} \\ &+\frac{\Psi_{i,j-1,k}+\Psi_{i,j+1,k}}{h_y^2} \\ &+\frac{\Psi_{i,j,k-1}+\Psi_{i,j,k+1}}{h_z^2}\end{aligned} \tag{79}$$

$$\Psi_{i,j,k} = \frac{\frac{\Psi_{i-1,j,k}+\Psi_{i+1,j,k}}{h_x^2}+\frac{\Psi_{i,j-1,k}+\Psi_{i,j+1,k}}{h_y^2}+\frac{\Psi_{i,j,k-1}+\Psi_{i,j,k+1}}{h_z^2}}{\frac{2}{h_x^2}+\frac{2}{h_y^2}+\frac{2}{h_z^2}+V_{i,j,k}-E} \tag{80}$$

Call the first time this is done step one. The solution at this lattice point is based on the value of its nearest neighbours. Then we move to the next lattice site and solve eq. 80 there, step 2. Of course, this changes the value of a parameter used in step 1, so that the solution of step 1 is no longer an exact solution. But the point is that it is getting closer. At any one time only the last lattice point visited is an exact solution to eq. 80, but we keep visiting each lattice site and updating the value there until the solution converges. Solving equation 80 assumes we already know the energy. Initially, however, we just guess the solution and the energy. So at the end of a loop through the entire lattice, the energy is updated. In the continuum, the energy would be given by

$$E = \sqrt{\frac{< H\Psi|H\Psi >}{< \Psi|\Psi >}} \tag{81}$$

Both the numerator and denominator can be discretized, and then a second order approximation to the integral is used (a 3-D trapezoid rule approximation) to sum up the contribution from each lattice site. This is done for both $H\Psi$ and Ψ and then put into equation 81. After visiting each lattice site, solving eq. 80 at each and then updating the energy according to eq. 81, we are ready to start the process over again, but this time we (hopefully) have a starting point that is closer to the solution. The change in energy is monitored (that is, when the energy is updated it is compared to its old value) and when the change is less than a certain tolerance parameter, the process is stopped. At the end of it all, Ψ is the approximate solution, and E is the associated energy.

2.3.2 Successive Over Relaxation

One of the problems with the Gauss-Seidel algorithm is that it takes a long time to converge. Successive over relaxation is a way of speeding things up. At each step in the solving process, instead of using eq. 80 to solve for the value at a lattice point, a slightly different equation is used.

$$\Psi_{SOR} = w * \Psi_{GS} + (1 - w)\Psi_{old} \tag{82}$$

where Ψ_{GS} is the solution calculated from the Gauss-Seidel algorithm, Ψ_{old} is the current value of Ψ, and Ψ_{SOR} is the new value.

Instead of moving just to the Gauss-Seidel value, the solution is set to go past it in an attempt to get closer to the solution faster. The parameter w in the equation is commonly taken to be 1.8, I have used 1.7 in my program after running tests and finding that it works well.

2.3.3 Extracting the scattering length from the 3d solution

Turning to the main goal of the problem, this section describes how to extract the scattering length from the Ψ given in 3 dimensions. As was shown before, the potential at large distances drops off to zero. Therefore, at large separation distances, the particles only feel the potential due to their own image charges. This situation can be solved with a 1-D solver as the only relevant dimension is the particle's y value. As described before, the energy can be extracted from this situation as well. Under the assumption that the particles do not affect each other at large distances (which is justified, since the potential goes to zero), the total energy can be written as the sum of the energy resulting from each particles interaction with its own images (E_y and E_z) and an energy coming from the particles interacting with each other (E_x). The energy found from successive over relaxation is the total energy, E_y and E_z can be found via the 1-d solver, so we are left with:

$$E_x = E_{total} - E_y - E_z \tag{83}$$

From this energy the scattering length can be extracted. As was shown in the 1-d case, the solution at large distances can be approximated by a sinusoidal form. It can be shown (see ref. [7]) that the energy of a sinusoidal wave is inversely proportional to $(L-a)^2$. In that case, the energy is given by:

$$E = \frac{C}{(L-a)^2}, \tag{84}$$

where L is the length of the superconductor, a is the scattering length, and C is a constant. The value of C was found by putting the known values of E, L, and a from the 1-D case into eq. 84.

In the 3-D case a similar method can be used. At large distances we can write the solution as a product of solutions depending on the different variables, and in that case the energies

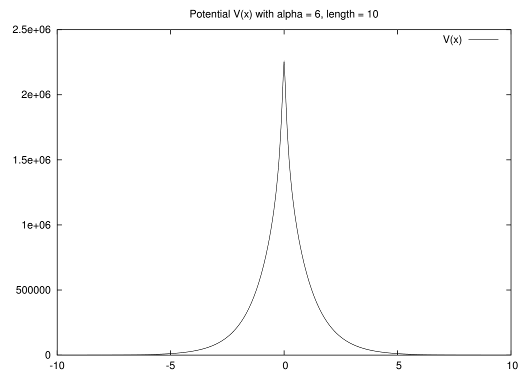


Figure 4: The potential energy function for parameters $\alpha = 6$, and $x_{max} = 10$. For values of x larger than this, the potential is approximately zero.

just add. Putting it all together, the successive over relaxation method returns E_{tot}, a 1-D solver can find E_y and E_z - from those E_x can be found. E_x is then of the form eq. 84, and the scattering length can be extracted.

3 Results and Analysis

The solution data are shown here. In each case plots of the wave function are shown and then the scattering length data is extracted and analyzed.

3.1 One Dimension

Let us first look at a plot of the potential function. This can be seen in fig. 4. Clearly the potential is only non-zero over a region approximately $-5 < x < 5$. Therefore, we only consider ranges larger than this. As it turns out, even $x_{max} = 10$ is not large enough.

3.1.1 Eigen-problem

The first area of exploration was to see if the solution was sinusoidal at large values of x. The following plots show the solution

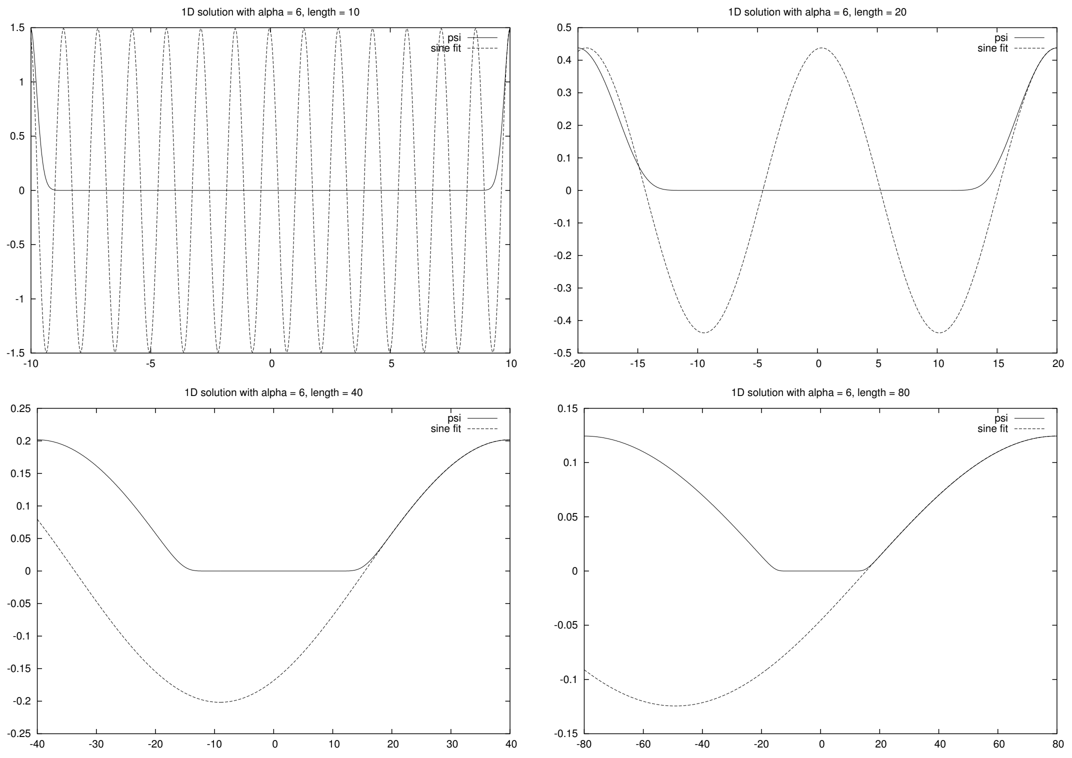


Figure 5: The solution and sine fit for length scales $x_{max} = 10$ (top left), 20 (top right), 40 (bottom left) and 80 (bottom right). All solutions used $\alpha = 6$. The sine wave fits well for the large x region, as expected. For $x_{max} < 20$, the sine wave oscillates rapidly, indicating that the solution still has a relatively high energy compared to the other solutions. The other interesting feature is that the solution goes to zero as $|x|$ gets smaller, indicative of the large potential in that region.

for various values of x_{max}, which is the length of the problem domain (see fig. 5). From looking at the plots, it is clear that we want to take x_{max} to be greater than 10 to get a solution with a small k (low energy).

Dr. Affleck has proposed that the scattering length a should be proportional to the logarithm of the potential multiplier. The multiplier is defined by

$$V_1 = 0.27 * 10^{\alpha} V_0(x) \tag{85}$$

where $V_0(x)$ is the potential that the fortran function which calculates the potential returns and $V_1(x)$ is the potential used

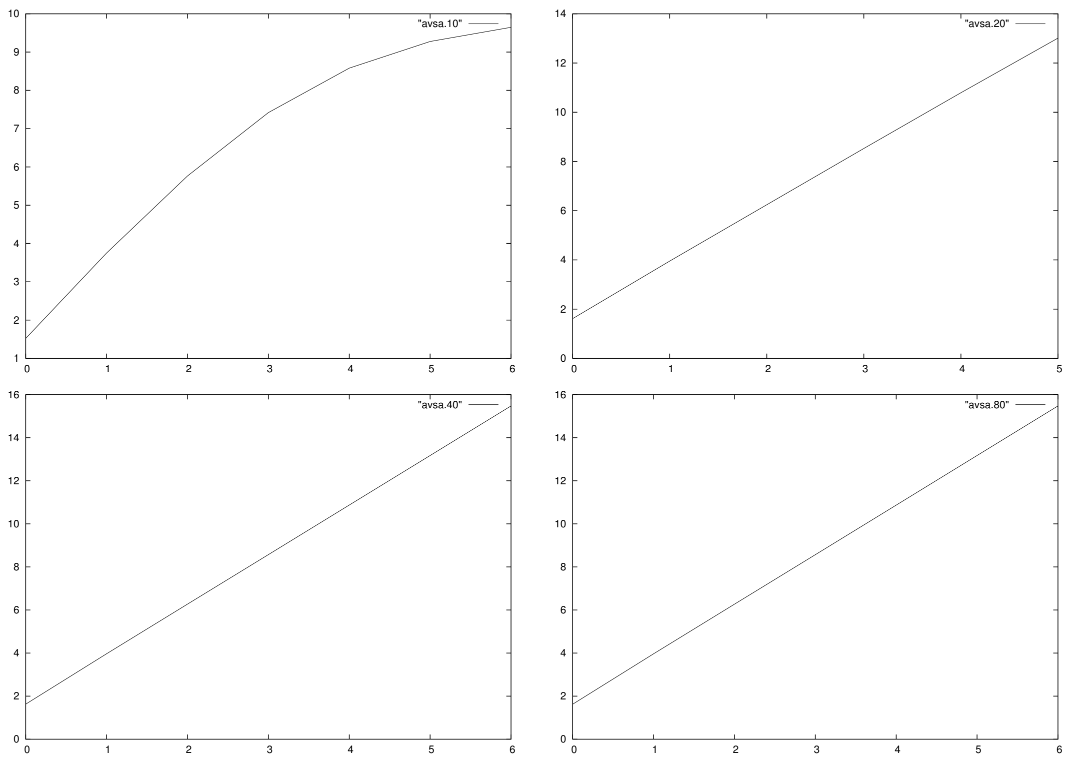


Figure 6: These are plots of the scattering length a versus the parameter α used in the potential calculation. The data is for different values of x_{max}. The top left is $x_{max} = 10$, the top right is $x_{max} = 20$, the bottom left is $x_{max} = 40$, and the bottom right is $x_{max} = 80$. These plots show further evidence for using a length greater than 10, as the data is not linear in that region.

in the solving of the problem. In this form, the hypothesis is that the scattering length is linearly proportional to α. We now investigate this hypothesis. The plot of the scattering length a versus α at various lengths is shown in figure 6, and it is indeed linear.

The next area of interest is looking at the scattering length as a function of the length x_{max}. It is expected that at a large enough value of x_{max}, the scattering length will approach a constant (taking the length to infinity implies letting the energy go to zero, which is the situation of interest). The data for this part of the experiment is shown in figure 7. It is clear that the scattering length does approach a constant value of ≈ 15.5.

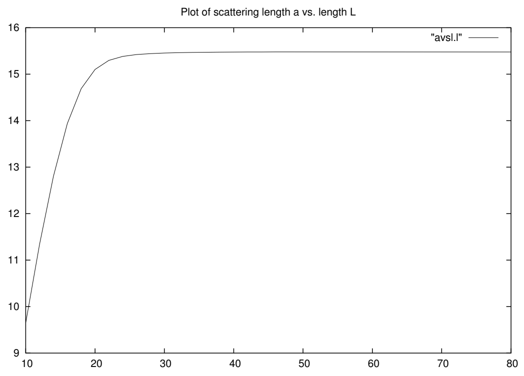


Figure 7: The plot of scattering length a versus length x_{max}. It is clear that the scattering length approaches a constant value, as expected. The value of the constant is ≈ 15.5.

3.1.2 Initial Value Problem

The IVP method was used as a way to validate the results found from the eigenvalue problem. There were numerical difficulties here, though, due to the large values of the potential function at small values of x. The initial value solver was not able to handle an α value greater than ≈ 1.5. As the value of α is supposed to be taken to 6, this is not much help. However, the solution is still explored for the small α region.

A plot of the solution is shown in fig. 8, along with a linear fit to the solution at large x. It is clear that the fit matches well. The ivp solver gives back values of y and the slope of y, so it is a simple matter to compute the constants in the linear equation:

$$y = k(x - a) \tag{86}$$

If the solver has just given back $y(xmax)$ and $y'(xmax)$ (the last data point), then the constants are found by:

$$k = \quad y'(xmax) \tag{87}$$

$$a = \quad xmax - \frac{y(xmax)}{y'(xmax)} \tag{88}$$

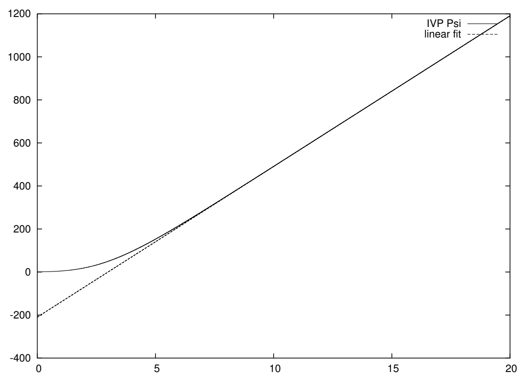


Figure 8: A plot of the solution of the ivp problem along with the linear fit.

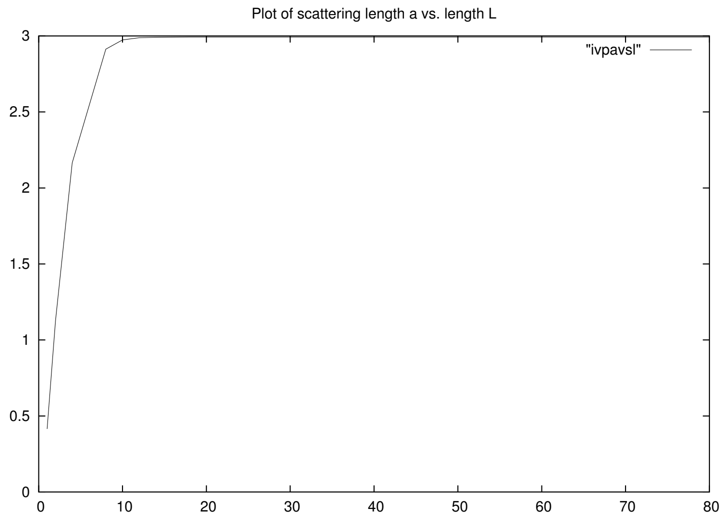


Figure 9: A plot of the scattering length a vs. length x_{max}. The value of α is 0.

This is how the scattering length is extracted from the ivp solution.

The next figure is of the scattering length a versus length x_{max} (fig. 9). The shape of the graph is the same as for the eigenvalue solution for a larger value of α, but the length scale is different. Here, the graph flattens out at $x \approx 10$, whereas in the eigenvalue solution it flattened out at $x \approx 25$. Also the constant value that they approach is different. These differences are due to the different values of α being used.

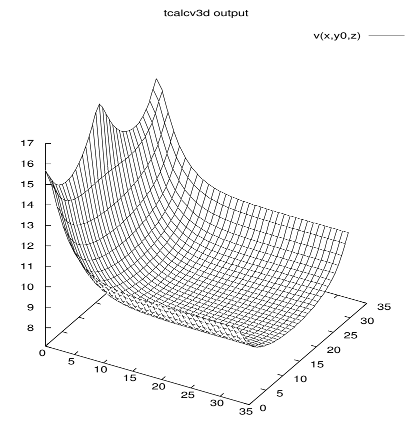


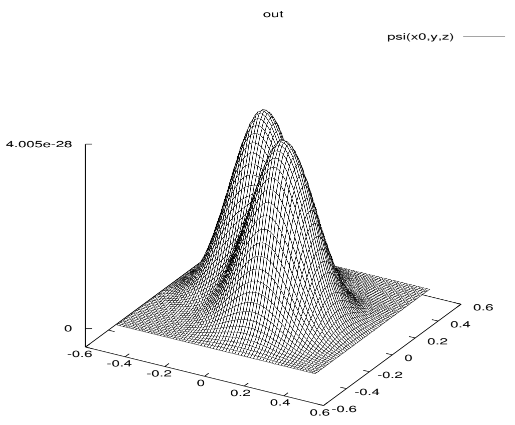
Figure 10: Plot of the potential function for the cross-section $y = 0$. The peak in the middle represents the repulsion from the other particle, while the peaks at the side represent the repulsion from the boundaries. The potential drops off as the particles move away from each other.

3.2 Three Dimensions

In the three dimensional problem, the particles are now allowed to move off the middle of the slab. The scattering length is then extracted from the energy as described in the theory section.

First let us look at the potential function. A three-dimensional cross section is shown in fig. 10. The potential has peaks where the particles overlap at $x = 0$ as in the one dimensional case, and also at the boundaries of y and z due to the repulsion from their image vortices.

A similar problem was encountered with the three dimensional problem as with the initial value method: the large coefficients that occur in the numerical approximations when α gets large make the algorithm fail. For values of α larger than 4, then solver did not converge. We proposed to find the scattering length for the lower values of α and then extrapolate the solution to the higher values. It was shown in the one dimensional case that the scattering length was linearly proportional to α; it is reasonable to think that this may be the case for three



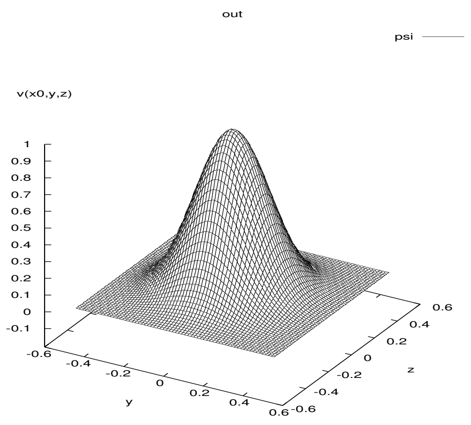


Figure 11: The plot on the left is a cross section of the solution at $x = 0$. This region is interesting because it is where the particles are repelled by each other as well as the boundaries. The two peaks indicate that there are two symmetrical positions at which the vortices are most likely to exist; it is where the repulsion from the boundaries balances the repulsion from the other vortex. The plot on the right shows a cross section taken at $x = x_{max}$, which is where the particles are furthest from each other. It clearly shows that both particles are most likely to be found on the horizontal axis when they are far apart. Note that the maximum value in this plot is ≈ 1 while for $x = 0$ it is of the order 10^{-28}, indicating that the probability of finding the vortices close together is almost 0.

dimensions as well.

The wave function has large peaks at $x = \pm x_{max}, y = z = 0$, as would be expected. A cross section taken at $x = x_{max}$ (that is when the vortices are furthest from each other) shows that the most probable location for both vortices is at $y = z = 0$ (see fig. 11). It is interesting to look at the solution for $x = 0$ though, because there the particles are repelled by each other as well as the slab boundaries. A plot of this is shown in fig. 11.

The next problem that occurred in solving the 3-D problem had to do with the resolution of the domain. As was described in the theory section, the total energy returned from the solving program is a sum of three energies: E_y, E_z, and E_x. The energies that result from the vortices being repelled by the boundaries (E_y and E_z) end up being orders of magnitude larger than E_x. As a result, the value of E_x found from eq. 83 is almost 0, and the accuracy of that value is not enough to be able to extract any meaningful data. This issue could theoretically be solved by increasing the resolution of the problem domain or by other methods described in the conclusion.

3.3 Convergence Tests

When solving a problem as complex as this, it is important to check that the answer is reasonable. Convergence testing is the act of testing to see that the solution has converged, ie. that the solution that was calculated is indeed a solution. It is similar to solving the matrix equation $Ax = b$ - when you calculate x, it is a good idea to check that Ax does in fact equal b.

The convergence testing done on the one dimensional problem is performed automatically in the graphics viewing program XVS. The solutions at resolution levels that are factors of 2 apart are inputs to the program. For example, one might use the solutions for $N = 64, 128$, and 256. The error is computed by taking the difference of the higher and lower order solutions (ie. the solution for $N = 128$ minus the solution for $N = 64$). If the solution is indeed converging, then the error from a higher resolution solution (ie. the solution for $N = 256$ minus the solution for $N = 128$) will be smaller by a factor corresponding to the increase in resolution (2, in this case). XVS multiplies the two graphs appropriately, such that if the solution is converging, then the graphs will overlap. This testing was done every step in finding the 1-D solution and it was confirmed that the solution was converging.

The method used to check convergence in the 3-D case is called an independent residual test. It relies on using a different numerical approximation scheme to find the same solution. In this case, a slightly different approximation to the second derivative was used. This independent residual approximation is applied to the solution that was found. Let the original Hamiltonian used in the solution be H and the independent residual Hamiltonian be H'. The solution is found by using H to find Ψ and E. Once they are found, we check that $H'\Psi - E\Psi$ goes to zero as the resolution gets higher.

4 Conclusion

The main goal of this project was to determine the scattering length a - a parameter that plays an important role in describing the properties of type II superconductors. For the 1-D case, this problem was solved and the value of a was found to be ≈ 15.5. For the 3-D case, time constraints and unforeseen numerical difficulties resulted in a solution that was not accurate enough to be able to extract a meaningful value for the scattering length. For future work, there are several possibilities to improve the accuracy. The most straightforward would be to simply increase the number of lattice points used in the solving program. Another solution would be to use higher order approximations in the program.

A secondary goal was to investigate whether or not the scattering length depended linearly on the parameter α. This problem was also solved and it was found that the scattering length is indeed linearly dependent on α.

5 Program Code

5.1 1D Eigenvalue Solver Code

```
c========================================================
      program symsolve
c     Author:   Brian Martin
c               PHYS 449:  Honours Thesis
c
c     last modified:  Apr. 25, 2005.
c
c     The program is used to solve the Schrodinger Equation
c     with a potential supplied by the file "calcv.f"
c========================================================
      implicit none
c     command line parsing tools
      integer i4arg, iargc
      real*8 r8arg
c     n becomes 2^level + 1, size of matrix.
c     maxn is the max size of the matrix
      integer level, n, maxn
      parameter (maxn = 2**12 + 1)
c     xmax is used in determing the solution range
      integer xmax,ymax
c     index variables
      integer i, j
c--------------------------------------------------------
c     dsyevx variables
c     see LAPACK documentation for complete details.
c--------------------------------------------------------
      integer lwork
      parameter (lwork = 2**24)
      real*8 a(maxn*maxn)
      real*8 vl, vu
      integer il,iu,m
      real*8 w(maxn)
      real*8 z(maxn*maxn)
      integer liwork
      parameter (liwork = 5*maxn)
      integer iwork(liwork)
      integer ifail(maxn)
      real*8 work(lwork)
      integer info

      real*8 psi(maxn)
      integer shape(1)
      real*8 bbox(2)
      real*8 alpha,lambyxi
      real*8 sinfit(maxn)
c--------------------------------------------------------
c     END OF VARIABLE DECLARATIONS
c========================================================

c--------------------------------------------------------
c     argument parsing
c--------------------------------------------------------
      if (iargc() .lt. 1) goto 900
c     set the size of the matrix
      level = i4arg(1,-1)
      if (level .lt. 1) go to 900
      n = 2**level + 1

c     make sure n is not too large
      if (n .gt. maxn) go to 900

c     get the range of eigenvectors to be found
c     the 1st one is the 0 eigenvalue vector, so we ignore
c     that
c     one and find the 2nd through (iu + 1) vector.
c     for example, an input value of iu = 2 finds the first
c     two non-zero eigenvalue solutions, which corresponds
c     to the 2nd and 3rd vectors.
      il = 1
      iu = 1
      if (iu .lt. 1 .or. iu .gt. n) go to 900
      alpha = r8arg(2,6.0d0)
      if (alpha .lt. 0.0d0) then
         write(0,*) 'alpha must be positive'
         goto 900
      end if

c     xmax is the range the solution is found on.  if not
c     supplied, it defaults to one.
      xmax = i4arg(3,10)
      ymax = i4arg(4,1)
      if (xmax .lt. 0 .or. ymax .lt. 0) then
         write(0,*) 'domain must be positive'
         goto 900
      end if
      lambyxi = r8arg(5,2.0d1)
c--------------------------------------------------------
c     end argument parsing
c--------------------------------------------------------

c--------------------------------------------------------
c     Main program.   all done in a driver subroutine
      call driver(n,a,vl,vu,il,iu,
     &     m,w,z,work,lwork,iwork,ifail,info,liwork,xmax,
     &     ymax,psi,shape,bbox,alpha,lambyxi,sinfit)
      stop
c========================================================
 900  continue
      write(0,*) 'usage: symsolve <n> '//
     &   '[<alpha> <xmax> <ymax> <lambyxi>]'
      stop
      end
c======================================================
c     END OF PROGRAM
c======================================================

      subroutine driver(n,a,vl,vu,il,iu,
     &     m,w,z,work,lwork,iwork,ifail,info,liwork,xmax,
     &     ymax,psi,shape,bbox,alpha,lambyxi,sinfit)

      implicit none
c======================================================
c     Variable declarations
c------------------------------------------------------
c     for argument parsing
      integer iargc
      integer i4arg

c     parameters for the matrix and vectors
c     and dgeev variables.
      integer    maxsize
      parameter (maxsize = 1000)

      integer n
      real*8 a(n, n)            ! hamiltonian matrix
      real*8 vl, vu
      integer m, il, iu
      real*8 w(n)
      real*8 z(n,n)

      integer liwork
      integer iwork(liwork)
      integer ifail(n)

      integer info, lwork
      real*8 work(lwork)

c     other necessary variables
      integer i, j, level, im1, ip1
      real*8 h, hm2, m2hm2
      real*8 x(n), v(n)

c      integer vmin, vmax

      integer    xmax,ymax
      real*8 potential

      logical evals, showpotential, analyze
      parameter (evals = .true.)
      parameter (showpotential = .false.)
      parameter (analyze = .true.)

      real*8 psi(n)
      real*8 e

      integer shape(1)
      real*8 bbox(2)

      real*8 alpha,lambyxi,fdp,fdpof,k,b,pi,al
      real*8 sinfit(n)
c---------------------------------------------------------
c     end of variable declarations
c=========================================================
      if (n .lt. 1) return

c---------------------------------------------------------
c     set up hamiltonian matrix
c---------------------------------------------------------

c     h is the step size
      h = 2.0d0 * xmax/(1.0d0*n-1.0d0)

c     x is the array of points used in
c     finding the potential v(x)
      do i = 1,n
         x(i) = (i-1.0d0)*h - xmax
      end do

      if (showpotential) then
         do i = 1, n
            call calcv(potential,x(i),1.0d-8,ymax,lambyxi)
            write(10,*) x(i), 0.27d6*potential
         end do
      stop
      end if

c     useful constants
      hm2 = -1.0d0/(h*h)
      m2hm2 = -2.0d0 * hm2

c      write(0,*) 'hs: ', h, hm2, m2hm2

c     the hamiltonian
      do i = 1,n
         im1 = i-1
         ip1 = i+1
         if (i .eq. 1) then
            im1 = n
         end if
         if (i .eq. n) then
            ip1 = 1
         end if
         a(i,im1) = hm2
         call calcv(potential,x(i),1.0d-20,ymax,lambyxi)
         a(i,i)  = m2hm2 + (0.27*10**(alpha))*potential
         a(i,ip1) = hm2
      end do

c==========================================================
c     find the eigenvalues and eigenvectors
c----------------------------------------------------------
      call dsyevx('V','I','U',n,a,n,vl,vu,il,iu,
     &     0.0d0,m,w,z,n,work,lwork,iwork,ifail,info)
c----------------------------------------------------------

c----------------------------------------------------------
c     display the result
c----------------------------------------------------------
      if(info .ne. 0) go to 800

      if (evals) then
         do i = 1,m
            write(0,*) i, w(i)
         end do
      end if

      i = 1
      do j = 1, n
         psi(j) = -z(j,i)
      end do

      call normalize(psi,n,h)

      do j = 1, n
         write(10,*) x(j),psi(j)
      end do
      e = w(i)

c     calculations to figure out parameters in the
c     sine fit
      fdp = (psi(n) - 2*psi(1) + psi(2))/(h**2)
      fdpof = -fdp/psi(1)
      k = sqrt(fdpof)
      b = psi(1)
      pi = acos(0.0d0)*2.0d0
      al = xmax - pi/(2*k)

      write(*,*)
      do j = 1, n
         sinfit(j) = b*sin(k*(x(j)-al))
         write(30,*) x(j), b*sin(k*(x(j)-al))
      end do

      shape(1) = n
      bbox(1) = -xmax
      bbox(2) = xmax

      call gft_out_bbox('sine',1.0d0,shape,1,bbox,sinfit)
      call gft_out_bbox('psi',1.0d0,shape,1,bbox,psi)

      write(20,*) alpha,xmax
      write(20,*) e,k,al

      call ires(psi,x,n,e,ymax,lambyxi)

      return

 800  continue
      write(0,*) 'full: dgeev() failed'
      stop

      end

c==========================================================
      subroutine normalize(psi, n, h)
      implicit none

      integer n,i
      real*8 psi(n)
      real*8 h
      real*8 magpsi

      magpsi = 0.0d0

      do i = 1, n
         magpsi = magpsi + ((psi(i))**2)*h
      end do
      magpsi = sqrt(magpsi)

      do i = 1, n
         psi(i) = psi(i) / magpsi
      end do

      magpsi = 0.0d0

      do i = 1, n
         magpsi = magpsi + ((psi(i))**2)*h
      end do
      magpsi = sqrt(magpsi)

      return
      end
c----------------------------------------------------------
c     routine to find the residual
      subroutine ires(psi,x,n,e,ymax,lambyxi)

      implicit none

      integer n
      real*8 psi(n)
      real*8 x(n)
      real*8 hm2,h

      integer i,il,ir

      real*8 res, v, e, length
      integer ymax
      real*8 lambyxi

      res = 0.0d0

      length = x(n) - x(1)
      h = length/(n-1)
      hm2 = -1.0d0/(h**2)
      do i = 1, n
         il = i-1
         ir = i+1

         if (i .eq. 1) then
            il = n
         end if
         if (i .eq. n) then
            ir = 1
         end if

         call calcv(v,x(i),1.0d-20,ymax,lambyxi)
         v = 0.27d6*v
         res = res+(hm2*(psi(ir)-2.0d0*psi(i)
     &        +psi(il))+(v-e)*psi(i))**2
      end do
      res = sqrt(res)/n

      write(0,*) 'res: ', res

      return
      end
c=========================================================
c     end of symsolve.f
c=========================================================

c-------------------------------------------------
c     Author:  Brian Martin
c     modified last:  Apr. 25, 2005.
c-------------------------------------------------
c-------------------------------------------------
c     a program to calculate the sum of
c     Bessel Functions.
c     The equation is:
c     V(x) = 2 *sum(n=-inf..inf)U(sqrt(x^2+(nw)^2))
c     Where U(r) is defined by:
c     U(r) = const. * Ko(r)
c     Ko is the 0th order bessel function
c-------------------------------------------------
      subroutine calcv(v,x,e,ymax,lambyxi)
      implicit none
c-------------------------------------------------
c     function declaration
      external u
      real*8 u
c-------------------------------------------------
c-------------------------------------------------
c     Variable declarations
c     x is separation distance.
c     v is the value of the potential that is returned
c     dv is the change in the potential when
c         adding new terms
c     e is the error tolerance
c     n is an index variable
c     lambyxi is the ratio lambda/xi
c     ymax represents the width of the slab
c-------------------------------------------------
      real*8 x, v, e, dv
      integer n
      real*8 lambyxi
      integer ymax
c-------------------------------------------------

c-------------------------------------------------
c     we initialize v to be the 0th element in
c     the sum, then, because the function is
c     symmetric, we double the sum from
c     1 to infinity
c-------------------------------------------------
      dv    = 2.0d0 * u(x,0,lambyxi)
      v     = dv
      n     = 1
      do while (abs(dv) > e)
         dv = 4.0d0 * u(x,n*ymax,lambyxi)
         n  = n+1
         v  = v + dv
      end do
      return
      end
c-------------------------------------------------
c     this is the function u, as defined above.
c-------------------------------------------------
      real*8 function  u(x,n,lambyxi)
        implicit none
c-------------------------------------------------
c     variables
c-------------------------------------------------
c     reals for the arbitrary constants and
c     the bessel function.
c-------------------------------------------------
        real*8 x, y
        real*8 k, lambda, xi
        parameter (k = 1.0d0)
        parameter (lambda = 1.0d0)

        real*8 dbesk0   !the bessel function
        integer n
        real*8 lambyxi

        xi = lambda/lambyxi
c     distance between vortices
        y = sqrt(x**2 + n**2)

        if (y .eq. 0.0d0) then
           u = log(lambda/xi)
           return
        end if

        u = k * dbesk0(y/lambda)
c     for y values greater than this, dbesk0 may underflow
c     but this approximation is valid since the function
c     is approaching 0.
        if (y < 32.0d0) then
           u = u - k * dbesk0(y/xi)
        end if
        return
      end
c=======================================================
c     end of calcv
c=======================================================
```

5.2 1D Initial Value Problem Code

Note that this code uses the same file “calcv.f” as used in the eigenvalue solver.

```
c=======================================================
c     ivp: Program which uses ODEPACK routine LSODA
c     to solve the second-order ODE
c
c        u''(x) - v(x)*u(x) = 0
c        0 <= x <= xmax
c
c     (' = d/dx), with initial conditions
c
c        u(0) = 1,   u'(0) = 0
c
c=======================================================
c     This program is based on 'tlsoda.f' which was
c     given by Matt Choptuik in Phys 410 at UBC.
c=======================================================
      program         ivp

      implicit        none
      include         'fcn.inc'

      real*8   calcv
      external calcv
      integer         iargc,        i4arg
      real*8          r8arg

      logical  option
      real*8   a
      integer  k, kmax
c---------------------------------------------------------
c     Command-line arguments:
c
c     xmax:    Final integration position
c     tol:     Error tolerance (this program uses LSODA's
c              pure absolute error control)
c     olevel:  Output level:  dtout = tmax/2**olevel
c---------------------------------------------------------
      real*8         xmax, u0, du0, tol
      integer        olevel
      real*8         r8_never
      parameter    ( r8_never = -1.0d-60 )

c---------------------------------------------------------
c     initial conditions
c---------------------------------------------------------
      parameter    ( u0 = 1.0d0 )
      parameter    ( du0 = 0.0d0 )
c---------------------------------------------------------

c=========================================================
c     Start of LSODA declarations
c=========================================================

c---------------------------------------------------------
c Note that 'fcn' and 'jac' are user supplied SUBROUTINES
c (not functions) which evaluate the RHSs of the ODEs and
c the Jacobian of the system.  Under normal operation,
c (as in this case), the Jacobian evaluator can be a
c 'dummy' routine; if and when needed, LSODA will compute
c a finite-difference approximation to the Jacobian.
c---------------------------------------------------------
      external       fcn,        jac
c---------------------------------------------------------
c     Number of ODEs (when written in canonical first order
c     form).
c---------------------------------------------------------
      integer         neq
      parameter     ( neq = 2 )
c---------------------------------------------------------
c     y(neq): Storage for approximate solution
c     t:   Initial time for LSODA integration sub-interval
c     tout:   Final time for LSODA integration sub-interval
c---------------------------------------------------------
      real*8         y(neq)
      real*8         x,          xout
c---------------------------------------------------------
c     Tolerance parameters:
c
c     The following comment block is extracted from the
c     LSODA documentation.
c---------------------------------------------------------
c rtol   = relative tolerance parameter (scalar).
c atol   = absolute tolerance parameter (scalar or array).
c   the estimated local error in y(i) will be controlled so
c   as to be less than
c        ewt(i) = rtol*abs(y(i)) + atol     if itol = 1, or
c        ewt(i) = rtol*abs(y(i)) + atol(i)  if itol = 2.
c   thus the local error test passes if, in each component,
c   either the absolute error is less than atol
c   (or atol(i)),
c   or the relative error is less than rtol.
c   use rtol = 0.0 for pure absolute error control, and
c   use atol = 0.0 (or atol(i) = 0.0) for pure relative
c   error
c   control.  CAUTION.. actual (global) errors may exceed
c   these local tolerances, so choose them CONSERVATIVELY.
c---------------------------------------------------------
      real*8         rtol,       atol
      integer        itol

c---------------------------------------------------------
c     Control parameters and return code (see below).
c---------------------------------------------------------
      integer        itask,      istate,     iopt

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

```
c     Work arrays.
c---------------------------------------------------------
      integer       lrw
      parameter    ( lrw = 22 + neq * 16 )
      real*8        rwork(lrw)

      integer       liw
      parameter    ( liw = 20 + neq )
      integer       iwork(liw)

c---------------------------------------------------------
c     'jt' defines which type of Jacobian is supplied or
c     computed; we use jt = 2 here which, as mentioned
c     above, instructs LSODA to compute a finite-difference
c     approximation to the Jacobian if and when needed.
c---------------------------------------------------------
      integer       jt

c=========================================================
c     End of LSODA declarations
c=========================================================

c---------------------------------------------------------
c     Miscellaneous variables
c---------------------------------------------------------
      real*8         dxout
      integer        it,       nxout


      real*8 ymax
      integer i
c---------------------------------------------------------
c     Argument parsing.
c---------------------------------------------------------
      if( iargc() .lt. 4 )  go to 900
      xmax   = r8arg(1,r8_never)
      tol    = r8arg(2,r8_never)
      olevel = i4arg(3,-1)
      w      = r8arg(4,r8_never)
      option = .false.
      kmax   = 1
      if( iargc() .eq. 5 ) then
         option = .true.
         kmax   = 100
      end if
      if( xmax .le. 0.0d0 ) then
         write(0,*) "xmax error: ", xmax
         goto 800
      end if
      if( tol .eq. r8_never) then
         write(0,*) "tol not read: ", tol
         goto 800
      end if
      if( olevel .lt. 0 ) then
         write(0,*) "olevel error: ", olevel
         goto 800
      end if
      if ( tol .gt. 10d-2 .or. tol .lt. 10d-12) then
         write(0,*) "tol out of range: ", tol
         goto 800
      end if
      if ( w .lt. 0.0d0 ) then
         write(0,*) "width less than 0: ", w
         goto 800
      end if

c---------------------------------------------------------
c     Set LSODA parameters ... see LSODA documentation
c     for fuller description.
c---------------------------------------------------------
      itol   = 1         ! Indicates that 'atol' is scalar
      rtol   = 0.0d0     ! Use pure absolute tolerance
      atol   = tol       ! Absolute tolerance
      itask  = 1         ! Normal computation
      iopt   = 0         ! Indicates no optional inputs
      jt     = 2         ! Jacobian type

c---------------------------------------------------------
c     Compute number of output times and output interval,
c     and intialize sub-interval start time and solution
c     estimate.
c---------------------------------------------------------
      nxout  = 2**olevel + 1
      dxout  = xmax / (nxout - 1)
      x      = 0.0d0
      y(1)   = u0
      y(2)   = du0


      ymax = 1.0d0
c-----------------------------------------------------------
c     Output initial solution.
c-----------------------------------------------------------
      write(*,*) x, y(1)/ymax

c-----------------------------------------------------------
c     Loop over requested output times ...
c
c     Set istate to 1 to indicate initial call, istate
c     should be set to 2 for subsequent calls, but lsoda
c     will automatically do this so long as the initial
c     call is successful.
c----------------------------------------------------------
      istate = 1


      do it = 2, nxout
c--------------------------------------------------------
c        Set final integration time for current interval
c---------------------------------------------------------
         xout = x + dxout
c---------------------------------------------------------
c        Call lsoda to integrate system on [t ... tout]
c
c        Note that LSODA replaces 't' with the value
c        of 'tout' if the integration is successful.
c--------------------------------------------------------

         call lsoda(fcn,neq,y,x,xout,
     &              itol,rtol,atol,itask,
     &              istate,iopt,rwork,lrw,iwork,liw,jac,jt)
c---------------------------------------------------------
c        Check return code and exit with error message if
c        there was trouble.
c---------------------------------------------------------
         if( istate .lt. 0 ) then
            write(0,1000) istate, it, nxout, x, x + dxout
1000        format(/' sode: Error return ',i2,
     &              ' from integrator LSODA.'/
     &              ' sode: At output time ',i5,' of ',i5/
     &              ' sode: Interval ',1p,e11.3,0p,
     &              ' .. ',1p,e11.3,0p/)
            go to 500
         end if
c------------------------------------------------------
c        Output the solution.
c-------------------------------------------------------
         write(*,*) x, (y(1)/ymax)
      end do


      write(0,*) y(1)
      a = xmax - y(1)/y(2)
      k = y(2)
      write(0,*) w, a
      write(0,*) 'k = ', y(2)
      do i = 1, nxout
         x = (i-1)*xmax/(nxout-1)
         write(10,*) x, k*(x - a)
      end do


 500  continue

      stop

 800  continue
         write(0,*) 'xmax > 0'
         write(0,*) '10e-12 <= tol <= 10e-2'
         write(0,*) 'olevel > 0'
         write(0,*) 'width > 0'
         write(0,*)


 900  continue
         write(0,*) 'usage: ivp <xmax> '//
     &              '<tol> <olevel> <width>'
      stop

      end

c=====================================================
c     Implements differential equations:
c
c     u''(x) = v(x)* u(x)
c
c     y(1) := u
c     y(2) := u'
c
c     y(1)' :=  y(2)
c     y(2)' := v(x) * y(1)
c
c     Called by ODEPACK routine LSODA.
c=====================================================
      subroutine fcn(neq,x,y,yprime)
         implicit    none
         include 'fcn.inc'

         integer     neq
         real*8      x,     y(neq),    yprime(neq)

         real*8      v

         call calcv(v,x,w,1.0d-8)

         yprime(1) =  y(2)
         yprime(2) =  v*y(1)

         return
      end

c====================================================
c     Implements Jacobian (optional).  Dummy routine in
c     this case.
c====================================================
      subroutine jac
         implicit    none

         return
      end
```

5.3 3D Solver Code

```
c==================================================
c     Author: Brian Martin
c     last modified: Apr. 25, 2005.
c
c     written for Undergraduate Honours thesis
c     University of British Columbia
c
      program solve3d
c
c     The program solves the 3-d Schodinger Equation
c     describing the interaction of two vortices in a
c     Type II superconducting slab.
c==================================================
      implicit none
c--------------------------------------------------
c     Variable Declarations
c---------------------------------------------------
c     for argument parsing
      integer iargc, i4arg
      real*8  r8arg
c     parameter for argument parsing
      real*8 r8never
      parameter (r8never = -1.0d-60)
c     psi is wavefunction
      integer maxsize
      parameter (maxsize = 257)
      real*8 psi(maxsize,maxsize,maxsize)
      real*8 psi2(maxsize,maxsize,maxsize)
c     v is the potential
      real*8 v(maxsize,maxsize,maxsize)
c     used for number of lattice points
      integer nx, ny, nz
c     length and width of the slab
      real*8  length, width
c     the output level.  xmax = 2^level + 1
```

```
      integer level
c     index variable
      integer i
c     the potential multipler ('alpha')
      real*8 amp
c     used for file output
      character*256 fname
c     the energy
      real*8 e
c-----------------------------------------------------
c     End of Variable Declarations
c     Start of program code.
c-----------------------------------------------------

c-----------------------------------------------------
c     Argument parsing.
c     Make sure there are the right number of arguments
c     and that those arguments have appropriate values.
c-------------------------------------------------------
      if (iargc() .lt. 4) go to 900
      length = r8arg(1,r8never)
      width  = r8arg(2,r8never)
      level  = i4arg(3,-1)
      amp    = r8arg(4,r8never)
      if (length .le. 0.0d0) then
         write(0,*) 'length must be greater than 0'
         write(0,*) 'length = ', length
         go to 900
      end if
      if (width .le. 0.0d0) then
         write(0,*) 'width must be greater than 0'
         write(0,*) 'width = ', width
         go to 900
      end if
      if (level .lt. 1) then
         write(0,*) 'level must be at least 1'
         write(0,*) 'level = ', level
         go to 900
      end if
      if (amp .lt. 0.0d0) then
         write(0,*) 'amp must be at least 0'
         write(0,*) 'amp = ', amp
         go to 900
      end if
      amp = 0.27d0*(10**amp)
c-------------------------------------------------------
c     End of argument parsing.
c--------------------------------------------------------

c--------------------------------------------------------
c     Set up lattice size.
c--------------------------------------------------------
      nx = 2**level + 1
      if (nx .gt. maxsize) then
         write(0,*) 'nx too large', nx, maxsize
         go to 900
      end if
      ny = 2**level + 1
      if (ny .gt. maxsize) then
         write(0,*) 'ny too large', ny, maxsize
         go to 900
      end if
      nz = ny

c------------------------------------------------------
c     Call the driver function, which does all the
c     real work.
c-----------------------------------------------------
      call initialize(psi,nx,ny,nz,e)
      call driver(psi,v,nx,ny,nz,length,width,fname,e,amp)
      call interpolate(psi,psi2,nx)
      nx = 2*nx-1
      ny = nx
      nz = ny
      call driver(psi2,v,nx,ny,nz,length,width,fname,e,amp)
      call interpolate(psi2,psi,nx)
      nx = 2*nx-1
      ny = nx
      nz = ny


      call driver(psi,v,nx,ny,nz,length,width,fname,e,amp)
      call interpolate(psi,psi2,nx)
      nx = 2*nx-1
      ny = nx
      nz = ny
      call driver(psi2,v,nx,ny,nz,length,width,fname,e,amp)

      call interpolate(psi2,psi,nx)
      nx = 2*nx-1
      ny = nx
      nz = ny

      call driver(psi,v,nx,ny,nz,length,width,fname,e,amp)
c     this function displays the relevent data
      call disp(psi,nx,ny,nz,length,width,fname,e,v)

      stop

c-----------------------------------------------------
c     Usage statement.
c-----------------------------------------------------
 900  continue
      write(0,*) 'usage: solve3d <length>'//
     &   <width> <level> <amp>'
      stop
      end
c=====================================================
c     END OF MAIN PROGRAM BLOCK.
c===================================================

c====================================================
c     Author: Brian Martin
c     last modified: Apr. 25, 2005.
c
c     written for Undergraduate Honours thesis
c     University of British Columbia
      subroutine initialize(psi,nx,ny,nz,e)
c     a subroutine used to make the initial guess
c     on the wavefunction.
c=====================================================
      implicit none

      integer nx,ny,nz
      real*8 psi(nx,ny,nz)
      real*8 e
      integer i,j,k
      e = 1.0d0
c-----------------------------------------------------
c     initialize faces to 0. (planes where y or z = +- w/2)
c     here and throughout the code, the index variables
c     correspond to the same dimension. i:x, j:y, k:z.
c-----------------------------------------------------
      do i = 1, nx
         do j = 1, ny
            do k = 1, nz
               psi(i,j,k) = 0.0d0
            end do
         end do
      end do
c-----------------------------------------------------
c     initial guess on psi and e
c     only need to set psi on interior since faces have
c     been set already and do not change.
c-----------------------------------------------------
      do i = 1, nx
         do j = 2, ny-1
            do k = 2, nz-1
               psi(i,j,k) = 1.0d0
            end do
         end do
      end do
      return
      end
c=====================================================
c     end of subroutine initialize
c=====================================================

c=====================================================
      subroutine driver(psi,v,nx,ny,nz,
     &         length,width,fname,e,amp)
c     driver subroutine which does all the work.
c=====================================================
      implicit none
c---------------------------------------------------
c     Variable declarations
```

```
c-----------------------------------------------------
c     number of lattice points
      integer nx,ny,nz
c     wave function
      real*8 psi(nx,ny,nz)
c     length and width of slab
      real*8 length, width
c     energy and change in energy
      real*8 e, de
c     index variables
      integer i,j,k,n
c     maxiter is maximum number of relaxation loops.
      integer maxiter
      parameter (maxiter = 100000)
c     step sizes
      real*8 hx, hy, hz
c     debugging variable
      logical checking
      parameter (checking = .false.)
      logical c2
      parameter (c2 = .true.)
c     convergence is a tolerance parameter.  when
c     de is less than convergence, the relaxation
c     is complete.
      real*8 convergence
      parameter (convergence = 1.0d-6)
c     the potential array
      real*8 v(nx,ny,nz)
      real*8 tempv
      real*8 x,y,z,xmax,ymin,zmin
      character*(*) fname
      real*8 error
      parameter (error = 1.0d-8)
      real*8 amp
c------------------------------------------------------
c     End of variable declarations.
c------------------------------------------------------
c------------------------------------------------------
c     Start of subroutine code.
c------------------------------------------------------
c------------------------------------------------------
c     first pass, set potential to 0.
c     initialize the potential array
c------------------------------------------------------
      xmax = length
      ymin = -width/2.0d0
      zmin = -width/2.0d0

c     set up step sizes
      hx = length/(nx-1)
      hy = width/(ny-1)
      hz = width/(nz-1)
      do i = 1, nx
         x = xmax - (i-1)*hx
         do j = 1, ny
            y = ymin + (j-1)*hy
            do k = 1, nz
               z = zmin + (k-1)*hz
               call calcv3d(tempv,x,y,z,width,error)
c     the constant value being used
c     assumes m_perp/m_z = 0.01
               v(i,j,k) = amp*tempv
            end do
         end do
      end do

      n = 0
c------------------------------------------------------
c     de is change in energy, we set it to be larger
c     than the convergence value because it will
c     default to 0, which means it will have already
c     converged.
c------------------------------------------------------
      de = 10.d0 * convergence
c------------------------------------------------------
c     Relaxation Loop.
c     While the change in energy is larger than the
c     convergence, update the wavefunction and energy.
c-----------------------------------------------------
      do while (abs(de) > convergence)
c------------------------------------------------------
c     Loop
c     solve is
c     designed to handle special boundary conditions
         do i = 1, nx
            do j = 2, ny-1
               do k = 2, nz-1
               call solve(psi,nx,ny,nz,i,j,k,hx,hy,hz,e,v)
               end do
            end do
         end do
c------------------------------------------------------
c------------------------------------------------------
c     Wavefunction has been updated, so now update
c     energy.
         call update(psi,nx,ny,nz,hx,hy,hz,e,de,v)
c------------------------------------------------------
c------------------------------------------------------
c     increment iteration counter
         n = n + 1
         if (c2) then
            if (mod(n,500) .eq. 0) then
               write(0,*) n
            end if
         end if
c------------------------------------------------------
      end do
c------------------------------------------------------
c     End of relaxation procedure.
c------------------------------------------------------
      write(0,*) 'energy = ', e, ', n = ', n
c------------------------------------------------------
c     normalize the wavefunction
      call normalize(psi,nx,ny,nz,length,width)
c------------------------------------------------------
      return

      end
c=====================================================
c     End of driver subroutine.
c=====================================================


c=====================================================
c     Author: Brian Martin
c     last modified: Apr. 25, 2005.
c
c     written for Undergraduate Honours thesis
c     University of British Columbia
c
      subroutine interpolate(psi,psi2,n)
c     this subroutine takes two arrays, one of twice
c     the dimension of the other, and interpolates
c     the values in between.
c=====================================================
      implicit none
      integer n
      real*8 psi(n,n,n)
      real*8 psi2(2*n-1, 2*n-1,2*n-1)
      integer i,j,k
      do i = 1, n
         do j = 1, n
            do k = 1, n
               psi2(2*i-1,2*j-1,2*k-1) = psi(i,j,k)
               if (i .lt. n) then
                  psi2(2*i,2*j-1,2*k-1)=
     &            0.5d0*(psi(i,j,k)+psi(i+1,j,k))
               end if
               if (j .lt. n) then
                  psi2(2*i-1,2*j,2*k-1)=
     &            0.5d0*(psi(i,j,k)+psi(i,j+1,k))
               end if
               if (k .lt. n) then
                  psi2(2*i-1,2*j-1,2*k)=
     &            0.5d0*(psi(i,j,k)+psi(i,j,k+1))
               end if
               if (i .lt. n .and. j .lt. n) then
                  psi2(2*i,2*j,2*k-1)=
     &                 0.25d0*(psi(i,j,k)+psi(i+1,j,k)+
     &                 psi(i,j+1,k) + psi(i+1,j+1,k))
               end if
               if (i .lt. n .and. k .lt. n) then
                  psi2(2*i,2*j-1,2*k)=
     &                 0.25d0*(psi(i,j,k)+psi(i+1,j,k)+
     &                 psi(i,j,k+1) + psi(i+1,j,k+1))
               end if
               if (j .lt. n .and. k .lt. n) then
```

```
                psi2(2*i-1,2*j,2*k)=
     &                0.25d0*(psi(i,j,k)+psi(i,j+1,k)+
     &                psi(i,j,k+1) + psi(i,j+1,k+1))
             end if
             if (i .lt. n .and. j .lt. n
     &           .and. k .lt. n) then
                psi2(2*i,2*j,2*k) =
     &                0.125d0*(psi(i,j,k)+psi(i,j,k+1)
     &                +psi(i,j+1,k)+psi(i,j+1,k+1)
     &                +psi(i+1,j,k)+psi(i+1,j,k+1)
     &                +psi(i+1,j+1,k)+psi(i+1,j+1,k+1))
             end if
          end do
       end do
      end do

      return
      end
c=====================================================
c     end of subroutine interpolate
c=====================================================

c=====================================================
c     Author: Brian Martin
c     last modified: Apr. 25, 2005.
c
c     written for Undergraduate Honours thesis
c     University of British Columbia
c
      subroutine calcv3d(v,x,y,z,w,e)
c
c     A subroutine used to calculate the potential
c     for the vortex interactions given the x,y, and z
c     positions of the vortices.
c======================================================
      implicit none
c-----------------------------------------------------
c     Variable Declarations
c------------------------------------------------------
c     input parameters:
c     v is outputed potential
c     x,y,z are the vortex positions
c     w is the width of the slab, e is the energy
      real*8 v,x,y,z,w,e
c     temporary variables used in calculations
      real*8 dv, temp
      integer n
      real*8 v1, v2, d, u
c     u is the function that calculates the potential
c     between any two charges, which is then summed up.
c------------------------------------------------------
c     End of Variable Declarations
c     Start of program code.
c------------------------------------------------------
      n = 1
      dv = 1.0d1*e
c     initial v to be 0th term in sum
      call getu(x,y,z,w,0,v)

c     add in term from y-z interaction
      d = sqrt(x**2 + (y - z)**2)
      v = v + u(d)

      do while (abs(dv) .gt. e)
         call getu(x,y,z,w,n,   v1)
         call getu(x,y,z,w,-1*n,v2)
         dv = v1 + v2
         v = v + dv
         n = n + 1
      end do

      return

      end
c------------------------------------------------
c------------------------------------------------

c------------------------------------------------
c------------------------------------------------
      subroutine getu(x,y,z,w,n,v)
      implicit none
      real*8 x,y,z,w,v
      integer n

      real*8 dy1,  dy2
      real*8 dz1,  dz2
      real*8 dyz1, dyz2
      real*8 dzy1, dzy2

      real*8 u1, u2

      real*8 u
c------------------------------------------------
c     there are four distances to find:
c     1. the distance from y to its images
c     2. the distance from z to its images
c     3. the distance from y to z images
c     4. the distance from z to y images
c
c     each of these has two terms which we break
c     note overcounting of interaction of y
c     with itself and z with itself (n=0 terms)
c------------------------------------------------

c     y with its images
      dy1 = abs(2*abs(y) - (2*n+1)*w)
      dy2 = abs(2*n*w)

c     z with its images
      dz1 = abs(2*abs(z) - (2*n+1)*w)
      dz2 = abs(2*n*w)

c     y with z images
      dyz1 = sqrt(x**2+(y+z-(2*n+1)*w)**2)
      dyz2 = sqrt(x**2+(y-z+2*n*w)**2)

c     z with y images
      dzy1 = sqrt(x**2+(y+z-(2*n+1)*w)**2)
      dzy2 = sqrt(x**2+(y-z+2*n*w)**2)

c     then calculate u for these terms and
c     find v
      if (n .ne. 0) then
         u1 = u(dy1)  + u(dy2)  + u(dz1)  + u(dz2)
         u2 = u(dyz1) + u(dyz2) + u(dzy1) + u(dzy2)
      else
         u1 = u(dy1)  + u(dz1)  + u(dz2)
         u2 = u(dyz1) + u(dzy1) + u(dzy2)
      end if

      v = u1 + u2

      return
      end
c------------------------------------------------
c------------------------------------------------

c------------------------------------------------
c------------------------------------------------
      real*8 function u(d)

      implicit none

      real*8 d

      real*8 lambda, xi
      parameter (lambda = 1.0d0)
      parameter (xi     = lambda/2.0d1)

      real*8 dbesk0

      if (d .eq. 0.0d0) then
         u = log(lambda/xi)
         return
      end if

      if (d .gt. 5.0d2) then
         u = 0.0d0
         return
      end if

      if (d/xi .gt. 5.0d2) then
         u = dbesk0(d/lambda)
         return
      end if
      u = dbesk0(d/lambda) - dbesk0(d/xi)
      return
      end
c======================================================
c     END OF CALCV3D.
c======================================================

c======================================================
      subroutine disp(psi,nx,ny,nz,length,width,fname,e,v)
c     routine to display the wavefunction.
c======================================================
      implicit none
c------------------------------------------------------
c     Variable declarations.
c------------------------------------------------------
c     lattice size
      integer nx,ny,nz
c     wavefunction
      real*8 psi(nx,ny,nz)
c     index variables
      integer i,j,k
c     display parameters
      integer shape(3)
      real*8  bbox(6)
      real*8 length
      real*8 width
      logical plot
      parameter (plot = .false.)
      real*8 e
      integer indlnb
      character*(*) fname
      integer uto, rc
      real*8 v(nx,ny,nz)
      real*8 x,y,z
c------------------------------------------------------
c     end of variable declarations.
c------------------------------------------------------

c------------------------------------------------------
c     Start of subroutine code.
c------------------------------------------------------
c------------------------------------------------------
c     set up display parameters.
      shape(1) = nx+1
      shape(2) = ny
      shape(3) = nz

      bbox(1) = -1.0d0
      bbox(2) = 1.0d0
      bbox(3) = -1.0d0
      bbox(4) = 1.0d0
      bbox(5) = -1.0d0
      bbox(6) = 1.0d0
c------------------------------------------------------

c------------------------------------------------------
c     this next part is used to output data to make 3-d
c     gnu-plots.
c------------------------------------------------------
      fname = 'out'
      uto = 1
      open(uto,file=fname(1:indlnb(fname)),
     &     form='formatted',iostat=rc)
      if (rc .ne. 0) then
         write(0,*) 'disp: Error opening ',
     &              fname(1:indlnb(fname))
         return
      end if

c     gnuplot output commands
      i=nx
      j = ny/2+1
      k = nz/2+1
      x = length/(2.0d0*(nx-1.0d0))*(2.0d0*i-nx-1.0d0)
      do j = 1, ny
         y = width/(2.0d0*(ny-1.0d0))*(2.0d0*j-ny-1.0d0)
         do k = 1, nz
            z = width/(2.0d0*(nz-1.0d0))*(2.0d0*k-nz-1.0d0)
            write(uto,*,iostat=rc) y,z,psi(i,j,k)
            if (rc .ne. 0) then
               write(0,*) 'disp:  error writing array'
            end if
         end do
         write(uto,*,iostat=rc)
      end do
      close(uto)

      fname = 'energy'
      uto = 2
      open(uto,file=fname(1:indlnb(fname)),
     &     form='formatted',iostat=rc)
      if (rc .ne. 0) then
         write(0,*) 'disp: Error opening ',
     &              fname(1:indlnb(fname))
         return
      end if

      write(uto,*,iostat=rc) e
      close(uto)

c     here is the 4-d plot
      call gft_out_bbox('psi', 1.0d0,shape,3,bbox,psi)
c------------------------------------------------------
      return
      end
c======================================================
c     End of display subroutine.
c======================================================

c======================================================
c     Author: Brian Martin
c     last modified: Apr. 25, 2005.
c
c     written for Undergraduate Honours thesis
c     University of British Columbia
      subroutine normalize(psi,nx,ny,nz,length,width)
c     a subroutine used to normalize the wavefunction.
c     It loops through and finds the magnitude of psi
c     and then divide each element by that value
c======================================================
      implicit none
      integer nx,ny,nz
      real*8 psi(nx,ny,nz)
      integer i,j,k
      real*8 mag_psi
      real*8 length, width
      real*8 hx,hy,hz
      real*8 max, pm

      hx = length/(nx-1)
      hy = width/(ny-1)
      hz = width/(nz-1)

      mag_psi = 0.0d0

      max = 0.0d0
      do i = 1, nx
         do j = 1, ny
            do k = 1, nz
               mag_psi = mag_psi + (psi(i,j,k)**2)*hx*hy*hz
               if (abs(psi(i,j,k)) .gt. abs(max)) then
                  max = psi(i,j,k)
               end if
            end do
         end do
      end do

      pm = 1.0d0
      if (max .lt. 0.0d0) then
         pm = -1.0d0
      end if

      do i = 1, nx
         do j = 1, ny
            do k = 1, nz
               psi(i,j,k) = psi(i,j,k) * pm / sqrt(mag_psi)
            end do
         end do
      end do

      return
      end
c======================================================
c     End of normalize subroutine.
c======================================================
c=====================================================
c     Author: Brian Martin
c     last modified: Apr. 25, 2005.
c
c     written for Undergraduate Honours thesis
c     University of British Columbia
      subroutine solve(psi,nx,ny,nz,i,j,k,hx,hy,hz,e,v)
c     routine to solve for psi(i,j,k)
c=====================================================
      implicit none
c-----------------------------------------------------
c     Variable declarations
c-----------------------------------------------------
c     lattice size
      integer nx, ny, nz
c     wavefunction
      real*8 psi(nx, ny, nz)
c     temporary variable used in SOR algorithm.
c     it is the gauss-seidel update value.
      real*8 psi_gs
c     the old psi value
      real*8 psi_old
c     the multiplier in the SOR algorithm
      real*8 mult
      parameter (mult = 1.7d0)
c     index variables
      integer i,j,k,il,ir
c     step sizes
      real*8 hx,hy,hz
c     energy
      real*8 e
c     potential
      real*8 v(nx,ny,nz)
c     temporary variables used to aid in calculations
      real*8 numx,numy,numz,den
c     debugging variable
      logical checking
      parameter (checking = .false.)
c-----------------------------------------------------
c     End of variable declarations.
c-----------------------------------------------------
c-----------------------------------------------------
c     Start of subroutine code.
c-----------------------------------------------------

c-----------------------------------------------------
c     calculation to update psi(i,j,k) based on its
c     nearest neighbors.
c-----------------------------------------------------
c     we need separate i indices to handle the special
c     x boundary conditions
      il = i-1
      ir = i+1
      if (i .eq. 1) then
         il = 2
         ir = 2
      end if
      if (i .eq. nx) then
         il = nx-1
         ir = nx-1
      end if

      numx = (psi(il,j,k) + psi(ir,j,k)) / hx**2
      numy = (psi(i,j-1,k) + psi(i,j+1,k)) / hy**2
      numz = (psi(i,j,k-1) + psi(i,j,k+1)) / hz**2

      den  = 2.0d0/hx**2 + 2.0d0/hy**2
     &     + 2.0d0/hz**2 + v(i,j,k) - e

      psi_gs = (numx + numy + numz) / den
      psi_old = psi(i,j,k)
      psi(i,j,k) = mult*psi_gs + (1.0d0 - mult)*psi_old

      return
      end
c=====================================================
c     End of solve subroutine.
c=====================================================

c=====================================================
c     Author: Brian Martin
c     last modified: Apr. 25, 2005.
c
c     written for Undergraduate Honours thesis
c     University of British Columbia
      subroutine update(psi,nx,ny,nz,hx,hy,hz,e,de,v)
c     routine to update the energy
c=====================================================
      implicit none
c-----------------------------------------------------
c     Variable declarations.
c-----------------------------------------------------
c     lattice size
      integer nx,ny,nz
c     wavefunction
      real*8 psi(nx,ny,nz)
c     step sizes
      real*8 hx,hy,hz
c     energy, change in energy
      real*8 e, de
c     potential
      real*8 v(nx,ny,nz)
c     variables to aid in calculation
      real*8 mag_psi,mag_psi_temp
      real*8 h_psi
      real*8 h_psi_temp
c     debugging variable.
      logical checking
      parameter (checking = .false.)
c     index variables
      integer i,j,k
      real*8 f
c-----------------------------------------------------
c     End of variable declarations.
c-----------------------------------------------------
c-----------------------------------------------------
c     Start of subroutine code.
c-----------------------------------------------------
c-----------------------------------------------------
c     This routine must loop through all lattice points
c     to calculate <H psi | H psi> and <psi | psi>.
c-----------------------------------------------------
      h_psi = 0.0d0
      mag_psi = 0.0d0
c-----------------------------------------------------
c     To figure out the desired quantities, we need to loop
c     over the entire space and look at nearest neighbors.
c     This region is the interior of the 3-d space plus
c     14 special cases:  the 8 corners and the 6 faces.
c     for these cases we use a linear continuation:
c         ie.  psi(i,0,k) = psi(i,2,k)
c-----------------------------------------------------
      do i = 2, nx-1
         do j = 2, ny-1
            do k = 2, nz-1
c-----------------------------------------------------
c     Calculation of <H psi | H psi>
c     using a 3-d trapezoid rule.
      h_psi_temp = f(psi,nx,ny,nz,v,i,  j,  k,  hx,hy,hz)
     &          + f(psi,nx,ny,nz,v,i  ,j,  k+1,hx,hy,hz)
     &          + f(psi,nx,ny,nz,v,i  ,j+1,k,  hx,hy,hz)
     &          + f(psi,nx,ny,nz,v,i  ,j+1,k+1,hx,hy,hz)
     &          + f(psi,nx,ny,nz,v,i+1,j,  k  ,hx,hy,hz)
     &          + f(psi,nx,ny,nz,v,i+1,j,  k+1,hx,hy,hz)
     &          + f(psi,nx,ny,nz,v,i+1,j+1,k  ,hx,hy,hz)
     &          + f(psi,nx,ny,nz,v,i+1,j+1,k+1,hx,hy,hz)
            h_psi_temp = h_psi_temp*(hx*hy*hz)/8.0

            h_psi = h_psi + h_psi_temp
c-----------------------------------------------------
c-----------------------------------------------------
c     Calculation of <psi | psi>
            mag_psi_temp = (hx*hy*hz/8.0d0)*
     &           ( (psi(i  ,j  ,k))**2 +
     &           (psi(i  ,j  ,k+1))**2
     &         +   (psi(i  ,j+1,k))**2 +
     &             (psi(i  ,j+1,k+1))**2
     &         +   (psi(i+1,j  ,k))**2 +
     &             (psi(i+1,j  ,k+1))**2
     &         +   (psi(i+1,j+1,k))**2 +
     &             (psi(i+1,j+1,k+1))**2)

            mag_psi = mag_psi + mag_psi_temp
c-----------------------------------------------------
            end do
         end do
      end do
c-------------------------------------------------------
c     update energy and change in energy
      de = sqrt(h_psi/mag_psi) - e
      e = sqrt(h_psi/mag_psi)
c-------------------------------------------------------
      return
      end
c=======================================================
c     end of update routine.
c=======================================================

c=======================================================
      real*8 function f(psi,nx,ny,nz,v,i,j,k,hx,hy,hz)
c     a function to find the midpoint between
c     lattice sites.
c=======================================================
      implicit none

      integer nx,ny,nz
      real*8 psi(nx,ny,nz)
      real*8 v(nx,ny,nz)
      integer i,j,k
      real*8 numx,numy,numz,fx,fy,fz,hx,hy,hz

      numx = - psi(i-1,j,k) - psi(i+1,j,k) + 2*psi(i,j,k)
      fx   = numx / hx**2
      numy = - psi(i,j-1,k) - psi(i,j+1,k) + 2*psi(i,j,k)
      fy   = numy / hy**2
      numz = - psi(i,j,k-1) - psi(i,j,k+1) + 2*psi(i,j,k)
      fz   = numz / hz**2
      f = fx + fy + fz + v(i,j,k)*psi(i,j,k)
      f = f**2
      return
      end
c=======================================================
c     end of function
c=======================================================
c=======================================================
c     End of update routine.
c=======================================================
```

6 Acknowledgements

Thank you to Dr. Affleck and Dr. Choptuik for their patience and wisdom. I also enjoyed several useful conversations with Roland Stevenson and Pal Sandhu.

References

[1] I. Affleck, W. Hofstetter, D.R. Nelson and U. Schollwoeck, J. Stat. Mech.: Theor. Exp. P10003 (2004).

[2] P.G. De Gennes, *Superconductivity of Metals and Alloys.* W.A. Benjamin, Inc., New York (1966).

[3] R.P. Feynman and A.R. Hibbs, *Quantum Mechanics and Path Integrals.* McGraw-Hill, New York (1965).

[4] D.V. Schroeder, *Thermal Physics.* Addison Wesley Longman, San Francisco, 2000.

[5] D.R. Nelson, *Vortex Entanglement in High-Tc Superconductors.* Phys. Rev. Lett. 60, 1973 (1988).

[6] D. Griffiths, *Introduction to Electricity and Magnetism.* Prentice Hall, New Jersey (1999).

[7] D. Griffiths, *Introduction to Quantum Mechanics.* Prentice Hall, New Jersey (1995).

[8] M.P.A Fisher, *Vortex-Glass Superconductivity: A Possible New Phase in Bulk High-Tc Oxides.* Phys. Rev. Lett. 62, 1415 (1989).