UNIVERSITY OF CALIFORNIA

Los Angeles

Semiclassical Modeling of Quantum-Mechanical Multiparticle Systems using Parallel Particle-In-Cell Methods

A dissertation submitted in partial satisfaction

of the requirements for the degree

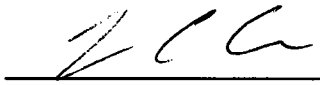
Doctor of Philosophy in Physics

by

Dean Edward Dauger

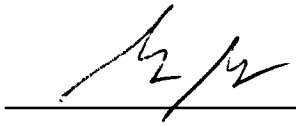
2001

The dissertation of Dean Edward Dauger is approved.

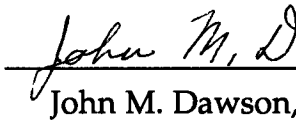


Steven C. Cowley

Warren B. Mori



Daniel X. Neuhauser



John M. Dawson, Committee Chair

University of California, Los Angeles

2001

To my parents, Alan and Marlene,

who were always fully supportive

of my work and my endeavors.

And Charlie and Allegra, my cats.

And to all those who have a vision,

a dream of something new and inspiring,

to express that Idea,

limited only by its own conclusion

if it has one.

Contents

List of Figures

List of Symbols

$\|\psi\rangle$	A Dirac ket of an arbitrary state ψ
$\langle\psi\|$	A Dirac bra of an arbitrary state ψ
$\langle\alpha\|\beta\rangle$	Inner product of a bra $\langle\alpha\|$ with a ket $\|\beta\rangle$
$\hat{x}$	Position operator
$\hat{p}$	Momentum operator
x	Space coordinate
p	Momentum coordinate
$\|x\rangle$	Ket of the position basis at position x
$\|p\rangle$	Ket of the momentum basis at momentum p
$\hat{H}$	Hamiltonian operator
π	The circumference of a circle divided by its diameter
i	Imaginary unity, defined by $i^2 = -1$
h	Planck's constant
$\hbar$	"h-bar", Planck's constant divided by two Pi
t	Time coordinate
Δt	The Quantum time step
∂t	The Classical time step
N	The number of classical time steps per quantum time step

$\frac{\partial}{\partial t}$	Partial derivative with respect to time
F	Force
m	Mass
a	Acceleration
$V(x)$	Effective potential as a function of position x
dx	Space differential
dp	Momentum differential
dt	Time differential
$\dot{x}$	Velocity
	"for all"
$\mathbf{1}$	Unity or identity operator
subscripts $_i$, $_j$	Arbitrary indices
l	Particle index
$\|\psi(t)\rangle$	Wavefunction $\|\psi\rangle$ at a time t
$\|\psi_0\rangle$	Initial wavefunction
$\|\psi_f\rangle$	Final wavefunction
$\cdots, \ddots, \vdots, \cdot\cdot$	Continuation symbols
$L(x_i, \dot{x}_i)$	The Lagrangian
S	Action, the time integral of the Lagrangian along a path

x_{cli}	Positions tracing a special path (a.k.a., "classical")
∂x_i	Variations off of the special path given by x_{cli}
$\frac{\partial^n}{\partial x^n}$	The nth partial derivative with respect to position x
S_{cl}	Classical Action
A	Temporary constant
S_2	Second-order and higher terms of the Action S
η	Column vector consisting of the variations ∂x_i
M	The semiclassical matrix
u	Constant part of the matrix M
w	Variable part of the matrix M
U, U^{-1}	An arbitrary unitary transformation and its inverse
M'	The matrix M in a basis of its own eigenvectors
η'	η in the basis of M'
m_i	The eigenvalues of M (and M')
$\det(M)$	The determinant of M
O(1), O(N)	"Of Order One", "Of Order N"
d_i	Determinant of the upper left $i \times i$ minor of M
$y(t)$ and $w(t)$	Function version of d_t and w_{t+1}, respectively
T	Period

$\left\lvert \frac{\partial p_{cl\,f}}{\partial p_0} \right\rvert$	Momentum Jacobian
$\lvert\Psi\rangle$	Total wavefunction
Δx	Grid (position) spacing
Δp	Momentum spacing
$p_{\max}$	Maximum sampled momentum
$\psi(x)$	$\langle x \lvert \psi \rangle$, The state $\lvert\psi\rangle$ in the position basis
$\psi(p)$	$\langle p \lvert \psi \rangle$, The state $\lvert\psi\rangle$ in the momentum basis
$\psi(x,t)$	$\langle x \lvert \psi(t) \rangle$, The state $\lvert\psi\rangle$ at time t in the position basis
q	Charge
ρ	Charge density
e	The magnitude of the charge of the electron
ϕ	Electric potential
$f(x_0, p_0)$	Temporary value dependent on x_0 and p_0
j, n	Integers
y_0, m	Temporary values
σ	Standard deviation
$\hat{A}$, $\hat{B}$	Dummy operators
$\langle \hat{A} \rangle$	Expectation value of $\hat{A}$
$\langle \hat{H} \rangle$	Energy of the state

$\langle (\Delta x)^2 \rangle$	Standard deviation in position
$\lvert\psi(x)\rvert^2$	Probability density of $\lvert\psi\rangle$ at position x
ω	Resonance frequency of a simple harmonic oscillator
c and $\tilde{c}$	Correlation function and its time Fourier transform
υ	Frequency
$\lvert n\rangle$	nth eigenstate of a system
L	Well width
E_n	Energy eigenvalue of $\lvert n\rangle$
ψ_{12}	Two-particle state
$\upsilon_{n,m}$	Frequency eigenvalue of a two-particle eigenstate
λ	One-dimensional atom potential gradient
$\lvert\tilde{\psi}(\upsilon)\rangle$	Time Fourier transform of a state $\lvert\psi(t)\rangle$
$\lvert e_n\rangle$	Arbitrary energy eigenstate
δ	Dirac delta function
k	Spatial wavenumber
E_L	Electric field, longitudinal component
$\frac{1}{8\pi}\lvert E_L(k,\omega)\rvert^2$	Electric field energy density
ε_L	Longitudinal permittivity

T	Temperature, in energy units
sec	second
cm	centimeter
eV , keV	electron-Volt, 1000 electron-Volts
ω_p	Angular plasma frequency
k_D	Debye wavenumber
v	velocity
η	arbitrary small constant
$f(v)$	velocity distribution of a plasma
z	unitless variable for algerbraic convenience
w	integration dummy variable
q	charge of a one-dimensional particle
n	three-dimensional number density
λ_{Debye}	Debye length
$\lambda_{deBrogli}$	de Broglie wavelength
$\|g\rangle$	Grid-point basis
$g(x)$	$\langle x\|g\rangle$, The grid-point function
$\tilde{g}(p)$	$\langle p\|g\rangle$, The grid-point function in momentum space
$\hat{H}_1$, $\hat{H}_2$	Hamiltonian operator for particle 1, 2
$\hat{H}_{12}$	Interaction part of the Hamiltonian operator

$\left[\hat{B},\hat{A}\right]$	$\hat{B}\hat{A}-\hat{A}\hat{B}$, Commutator of $\hat{B}$ with $\hat{A}$
V_{1eff}	Effective potential for particle 1

Acknowledgements

This work was funded by the Institute for Scientific Computing Research at Lawrence Livermore National Laboratory (LLNL / ISCR Contract Numbers: 00-08, 99-012, and 98-09) in a collaboration with Dr. Dennis Hewett.

Credit goes to Jose Louis Hales-Garcia and Jan de Leeuw of the Department of Statistics at the University of California, Los Angeles, for their permission and generous contributions of computational time and assistance in using their gSCAD cluster.

The computing platform for this work was entirely Macintosh.

Thanks goes to Dr. Alan B. Dauger for his advice and encouragement, to Robert M. Zirpoli III for his support during the hard times, and Catherine C. Venturini for her help, encouragement, and advice in editing this dissertation. Thanks to Dr. Warren B. Mori for his support, saving the day at the right moment, and more than once.

And special thanks goes to Dr. John M. Dawson and Dr. Viktor K. Decyk for their invaluable advice, generous support, and gracious understanding throughout the course of this work. Their patience and the freedom they allowed cultivated the fullest exploration of creativity and ideas of all kinds.

VITA

August 27, 1972	Born, Huntington Beach, California, USA
1992-1994	Software Engineer HSC Software, Inc. Santa Monica, California
1994	B. S., Mathematical Physics Harvey Mudd College Claremont, California
1994	Technical Support Engineer Jet Propulsion Laboratory Pasadena, California
1995	Jon A. Wunderlich Prize Harvey Mudd College Claremont, California
1996	M. S., Physics University of California Los Angeles, California
1995-1997	Teaching Assistant Department of Physics University of California, Los Angeles
1998	*Fresnel Diffraction Explorer* Student Winner, Eighth Educational Software Contest *Computers In Physics*, American Institute of Physics College Park, Maryland
1999	*Atom in a Box* Student Winner, Ninth Educational Software Contest *Computers In Physics*, American Institute of Physics College Park, Maryland
1998-2000	Software Developer Project AppleSeed Department of Physics University of California, Los Angeles
1995-2000	Graduate Researcher Department of Physics University of California, Los Angeles

PUBLICATIONS

D. E. Dauger, “Simulation and study of Fresnel diffraction for arbitrary two-dimensional apertures”, *Computers In Physics*, **10** (6), p. 591, (1996).

V. K. Decyk, D. E. Dauger, P. R. Kokelaar, “How to Build an AppleSeed: A Parallel Macintosh Cluster for Numerically Intensive Computing,” *Physica Scripta*, **T84**, p. 85-88, (2000).

ABSTRACT OF THE DISSERTATION

Semiclassical Modeling of Quantum-Mechanical Multiparticle Systems using Parallel Particle-In-Cell Methods

by

Dean Edward Dauger

Doctor of Philosophy in Physics

University of California, Los Angeles, 2001

Professor John M. Dawson, Chair

We are successful in building a code that models many particle dynamic quantum systems by combining a semiclassical approximation of Feynman path integrals with parallel computing techniques (particle-in-cell) and numerical methods developed for

simulating plasmas, establishing this approach as a viable technique for multiparticle time-dependent quantum mechanics. Run on high-performance parallel computers, this code applies semiclassical methods to simulate the time evolution of wavefunctions of many particles. We describe the analytical derivation and computational implementation of these techniques in detail. We present a study to thoroughly demonstrate the code's fidelity to quantum mechanics, resulting in innovative visualization and analysis techniques. We introduce and exhibit a method to address fermion particle statistics. We present studies of two quantum-mechanical problems: a two-electron, one-dimensional atom, resulting in high-quality extractions of one- and two-electron eigenstates, and electrostatic quasi-modes due to quantum effects in a hot electron plasma, relevant for predictions about stellar evolution. We supply discussions of alternative derivations, alternative implementations of the derivations, and an exploration of their consequences. Source code is shown throughout this dissertation. Finally, we present an extensive discussion of applications and extrapolations of this work, with suggestions for future direction.

I. Introduction

A. Motivation

Quantum mechanics is one of the most significant scientific developments in the twentieth century. The theory is also one of the most controversial, because it deals directly with phenomena of the universe that are not easily accessible by unaided human perception, yet at the same time it provides the essential answers that explain much of what we see around us and make our existence possible. Some of the consequences of the theory, taken to their logical extreme, seem to defy a resemblance to reality, yet, by exercising the proper patience with the concepts, one can find that quantum mechanics possesses an internally consistent logic all its own and indeed has connections to our normal perceptions. Further, one may discover familiar pieces embedded in a world outwardly unfamiliar.

Scientists of the early twentieth century were attempting to grapple with the fundamental nature of matter and light. While scientists at the time were comfortable with a wave theory of light, derived from Maxwell's equations, Planck's theory of black body radiation, introduced in 1900,[1] suggested that light could behave in discrete units called *quanta*. In 1905, Einstein[2] extended on this idea to explain the photoelectric effect, which became additional evidence for light showing particle behavior.

While scientists were comfortable with a particle model of matter, de Broglie suggested, in his 1924 thesis[3], that matter has wave properties. In analogy to the phase of light quanta traveling along light rays, he proposed that a particle of matter gains phase as it travels. Inspired by the similarity between Fermat's principle and the principle of least action, he identified matter's phase with *the classical action*, the integral of the Lagrangian, *along the particle's path.* Reinterpreting Planck's and Einstein's light quantization rule instead for matter, he then applied this idea to create a model of the atom that quantitatively and conceptually explained Bohr's earlier model. De Broglie's model predicted that electrons will only be stable in particular orbits around a nucleus because the electron's phase constructively interferes, resonating like a standing wave, on the orbit's *path.* Seeing light, normally accepted as waves, behave like particles allowed many to consider the notion that matter, normally accepted as particles, could behave like waves. The rules governing this

behavior came to be known as *quantum mechanics*.

In 1926, Schrödinger published a series of papers [4] on wave mechanics, introducing a differential equation satisfying the de Broglie matter-wave model. In the same year, the WKB method [5] was developed to help find approximate solutions to Schrödinger's equation. In 1928, Van Vleck [6] generalized the WKB method to higher dimensions and derived the appearance of the classical action in a complex exponential, to be later identified as a propagator. This work was among the earliest to show connections between classical mechanics and quantum mechanics.

Inspired by discoveries of Dirac [7], Feynman published his seminal paper [8] on path integrals in 1948. The evolution of a particle could be thought to be a sum over possible paths whose contributions are described by a propagator. This paper was significant because it demonstrated explicitly how Feynman's rigorous form of path integration could be used to derive quantum mechanics, clearly establishing the technique's relevance as a method alternative to that of Schrödinger. In fact, Feynman's path integral was a more direct application and generalization of de Broglie's idea than Schrödinger's equation. Also based on Dirac's work, he showed how, in typical cases, a sum over these paths through space could be seen to simplify to a sum of classical paths. The familiar arising from the unfamiliar, classical dynamics was seen to arise out of a purely quantum-mechanical concept, providing a clear connection between classical

and quantum theory. The term *semiclassical* was later coined for this apparent merge of classical and quantum ideas.

Feynman later built on his path integral work.[9] In 1967, Gutzwiller[10] used Feynman's path integrals to rederive Van Vleck's propagator with the addition of phase corrections due to caustics along periodic orbits. These caustics were identified by properties of the eigenvalues of the semiclassical matrix, used in the determinant factor that expresses focusing in the application of the WKB-like methods to semiclassical paths.

In the early 1990s, Heller and Tomosovic produced a series of articles[11-15] demonstrating accuracy and stability of quantum-mechanical calculations using long classical paths based on the formula of Van Vleck, Maslov[16], and Gutzwiller. Some of the techniques built upon the developments of many others.[17] These and related work[18,19] provided evidence, at least for single particle cases, for the computational viability of using many classical paths to answer specific questions about quantum-mechanical systems, including those that are chaotic.

Meanwhile, plasma physics has developed significantly in the last half of the twentieth century. Plasmas, by definition, are collections of particles under the influence of their mutual electromagnetic fields and following paths determined by classical mechanics. Buneman and Dawson developed the earliest computational models of plasmas.[20-23] These systems were one-

dimensional "sheet models" of the plasma, and efficient computational techniques for such models were developed. [24] Later, these models were extended to two and three dimensions by introducing methods to efficiently solve for electrostatic fields and combining the use of grid points [25] with the application of a Fast Fourier Transform (FFT) algorithm [26] to solving Poisson's equation in Fourier space [27]. These developments made efficient modeling of multidimensional plasmas possible.

Further improvements in plasma modeling came in step with the evolution of computational hardware. In particular, Particle-In-Cell (PIC) techniques to model plasmas on parallel computing hardware has seen great strides in work by Dawson, Decyk, and others. [27-33] Such plasma PIC simulations effectively and efficiently utilize such computational resources, achieving 90% parallelism and 40% of estimated peak hardware speed. In the 1990s, problems involving up to 2×10^8 particles on 32×10^6 grid points in three dimensions have become possible. These methods are shown to be robust and portable [34,35], and have run successfully on a wide range of computers (e.g., Cray-90s, T3Ds, T3Es, SGIs, IBM SP2s, and Macintosh clusters [36]).

Dawson, familiar with the efficiency of these plasma methods to manage particles and calculate their classical paths, conceived of the idea to apply these techniques to the classical paths in the semiclassical methods referred to by Heller and Tomsovic [13]. If we assume thousands of classical paths could be

used to evolve a system of one quantum particle, then could millions of classical paths be used to evolve a system of hundreds, or perhaps thousands, of quantum particles?

If successful, such a code could model scores of phenomena where quantum effects are important and answer some of the most difficult questions involving quantum mechanics. This modeling method would allow a detailed investigation of optical properties, ionization potential, conductance, and a host of other experimentally determined material properties. This tool could be used for the design and physical understanding of devices where quantum mechanics is important. Ultimately, with the incorporation of multiple dimensions, spin phenomena, and electromagnetism, this method would be able to model atoms, chemical reactions, quantum electronics, solid-state physics, and a multitude of other addressable physical problems. Cross-pollinated from plasma computation and semiclassical and quantum theory, this idea and its potential implications are the motivation of this work.

B. Existing Methods

Other methods that address quantum behavior exist, varying in complexity and accuracy. Among the candidates are mean-field methods and their extensions applied to solutions determined using the Schrödinger

equation, usually in a finite-difference or FFT form.[37-39] Other methods exist that approximate the particles as Gaussian wavepackets. Some use a "frozen" Gaussians, i.e., those of fixed width,[40] to evolve a wavefunction, while others use Gaussians with parameters that change in response to the system.[41]

Finally, many applications of semiclassical methods and their derivatives have been accomplished. These are usually directed at particular properties of a quantum system, most commonly the energy spectrum, using a wide variety of approaches.[11,13,15,17,19,42-50] Some have met with great success, and some are limited in quality for long time scales. The author finds the reference by Schulman[17] to continue to be an excellent authority on path integration, while other references[51] reflect more recent work.

Computational application of semiclassical methods most commonly use the Van Vleck-Gutzwiller-Maslov propagator. For example, based on work by Heller[52], Simontti et al[53] have developed clever methods for solving for time-independent eigenstates of two-dimensional billiard-type quantum systems. They focus on constructing the eigenstate data at the boundary of the system using a superposition of plane waves determined by segments of periodic classical orbits they locate in the system. Their methods use the Van Vleck-Gutzwiller-Maslov propagator to determine relevant properties of these periodic orbits. They then use Green's theorem to derive the interior of the eigenstate using the boundary information. Other work on time-dependent

propagation of wavefunctions using classical paths and that propagator are rare and meet with limited success. [11]

To the author's knowledge, the work presented in this dissertation is the first to directly use classical paths to accurately propagate time-dependent quantum wavefunctions. In this work, the author derives a propagator directly from basic quantum mechanics and Feynman path integrals. This propagator is designed for the computational time-dependent evolution of dynamic discretized wavefunctions. Its derivation is guided by the form of the Van Vleck propagator, Gutzwiller's work, and a section of Chapter 14 of Schulman [17]. Otherwise, this propagator, its development, implementation, study, and application are new and not found in previous literature. This work is also the first to use these methods to simulate the dynamics of many (i.e., hundreds) mutually interacting quantum wavefunctions.

C. Outline

This dissertation is an exposition of methods used to combine the semiclassical methods for solving quantum-mechanical problems with computational techniques from plasma PIC simulations for implementation on parallel computers. It is meant to serve as a guide for future use and development of both the existing quantum PIC code and any future codes

using similar techniques. These chapters show that this work contains a highly unique blend of quantum mechanics, classical mechanics, integration methods, numerical methods, parallel computing techniques, and verification and validation techniques.

Chapter II provides the theoretical foundation of the methods used to evolve quantum-mechanical wavefunctions. Chapter III describes how these equations were implemented in the actual code, defining and presenting each of its parts. Chapter IV presents and builds on a study, using basic quantum mechanics, applied to validate the quantum PIC code and demonstrate its capabilities. Chapter V presents an application of the quantum PIC code to the one-dimensional atom, and Chapter VI shows the code's application to energy fluctuations in a plasma. Chapter VII suggests the future possibilities of this code and others like it. Appendix A describes ideas, and their consequences, that were developed in the course of the research that led to the solution presented here. Appendix B and C provide key portions of the source code of the quantum PIC code, the code used to visualize the results, and other related code.

D. Conventions

The convention used in this presentation uses the Dirac bra-ket notation

($|\psi\rangle$) to represent wavefunctions. The position operator $\hat{x}$ has an associated complete position basis set $\{|x\rangle\}$, and its dual is the momentum operator $\hat{p}$ with its complete momentum basis set $\{|p\rangle\}$. These spaces are related through the Fourier transform kernel, $\langle x|p\rangle = \frac{1}{\sqrt{h}} \exp(\frac{2\pi i xp}{h})$, where h is Planck's constant. The time-dependent Schrödinger equation is $\hat{H}|\psi\rangle = i\hbar\frac{\partial}{\partial t}|\psi\rangle$, where $\hat{H}$ is the Hamiltonian operator and $\hbar \equiv h/2\pi$. This convention is best expressed in a reference by Townsend.[54]

II. Theory

A. The Approach

Our approach to evolving a set of quantum-mechanical wavefunctions is the following: Each wavefunction can be evolved using a large number of arbitrary paths. Because of the nature of the contributions of these paths, the total contribution can be simplified to just those from the classical paths. These contributions form the wavefunction at the new time step. Duplicating this procedure for all wavefunctions updates the entire system to the new time step, allowing the process to repeat.

We begin with the paths used for Feynman path integrals.[8] More commonly used in quantum field theory, these paths begin at an initial position in the wavefunction at the earlier time step, weave their way through space, and end at a final position in the wavefunction at the later time step. The

contribution of this path is the product of the wavefunction evaluated at the beginning of the path and a complex number whose phase is proportional to the action, the integral of the Lagrangian, along that path. These contributions are summed to form the new wavefunction.

The technique used to simplify the contributions to the classical paths is called a “semiclassical approximation”. Although not exactly identical, it has much in common with WKB techniques and stationary-phase methods. It involves summing the contributions from paths with the same initial and final positions. The result is that the paths in the vicinity of the path whose action is an extremum provide the most significant contributions. The property of these paths to focus on is their phase. The phase difference between paths changes as a function of their variation off the extremum path. In part because Planck’s constant is so small, it tends to be the case that this phase difference increases quickly with variation. This property is essential to this approximation. Its key is in showing that this rapid variation in phase causes their contributions to cancel each other. This cancellation dominates over all other effects. The special path with the extremum action, also found using the Lagrangian-based calculus of variations of classical mechanics, is called the classical path.

In the following sections, we will show derivations of the semiclassical methods, from their start in basic quantum mechanics to the complete contributions of the classical paths given by the semiclassical approximation.

We show these derivations because of two problems found in the course of this work: 1. Such calculations could not be found together in detail in any other source. 2. Previous results (such as the Van Vleck-Gutzwiller-Maslov [19] propagator) were found to be inappropriate to this application. To overcome these difficulties, the author reconstructed the semiclassical derivations from basic quantum theory and customized them for this dissertation's application, resulting in a new technique not found in previous literature. In the context of quantum field theory, *virtual particles* are said to follow the paths forming a Feynman path integral. Likewise, we coin the term "*virtual classical particles*", which trace the classical paths in this discussion.

B. Feynman Path Integrals

The theoretical basis for the quantum-mechanical methods used here is the Feynman path integral. We begin with a result of the time-dependent Schrödinger equation, which will allow us to derive a precise Feynman path integral more quickly. Consider the time evolution of one wavefunction, $|\psi\rangle$, over an interval from t to $t + \Delta t$,

$$|\psi(t + \Delta t)\rangle = \exp(-\frac{i\hat{H}\Delta t}{\hbar})|\psi(t)\rangle \qquad (1)$$

where $\hbar$ is Planck's constant divided by 2π, and $\hat{H}$ is the complete Hamiltonian,

$$\hat{H} = \sum_l \frac{\hat{p}_l^2}{2m} + \sum_l V_l(\hat{x}_l) \tag{2}$$

where V_l is the effective potential encountered by particle l. Define $|\psi_f\rangle \equiv |\psi(t+\Delta t)\rangle$ and $|\psi_0\rangle \equiv |\psi(t)\rangle$. We then divide this time interval into N intervals, each spaced by ∂t_i:

$$\begin{aligned} |\psi_f\rangle = \exp(-\frac{i\hat{H}\partial t_N}{\hbar})\exp(-\frac{i\hat{H}\partial t_{N-1}}{\hbar})\cdots\exp(-\frac{i\hat{H}\partial t_{i+1}}{\hbar})\exp(-\frac{i\hat{H}\partial t_i}{\hbar})\cdots \\ \cdots\exp(-\frac{i\hat{H}\partial t_2}{\hbar})\exp(-\frac{i\hat{H}\partial t_1}{\hbar})|\psi_0\rangle \end{aligned} \tag{3}$$

such that

$$\sum_{i=1}^{N} \partial t_i = \Delta t \text{ and } \partial t_i > 0, \quad i. \tag{4}$$

Next, insert $\mathbf{1} = \int dx_i |x_i\rangle\langle x_i|$, for $0 \le i \le N$, in between the exponentials:

$$\begin{aligned} |\psi_f\rangle = \int \prod_{j=0}^{N} dx_j |x_N\rangle\langle x_N| \exp(-\frac{i\hat{H}\partial t_N}{\hbar})|x_{N-1}\rangle\langle x_{N-1}|\exp(-\frac{i\hat{H}\partial t_{N-1}}{\hbar})|x_{N-2}\rangle\cdots \\ \cdots\langle x_i|\exp(-\frac{i\hat{H}\partial t_i}{\hbar})|x_{i-1}\rangle\cdots\langle x_1|\exp(-\frac{i\hat{H}\partial t_1}{\hbar})|x_0\rangle\langle x_0|\psi_0\rangle \end{aligned} \tag{5}$$

This is an N+1-dimensional integral.

Consider the ith term, for $1 \le i \le N$, of the above product. Insert

$\mathbf{1} = \int dp_i |p_i\rangle\langle p_i|$:

$$\langle x_i|\exp(-\frac{i\hat{H}\partial t_i}{\hbar})|x_{i-1}\rangle = \int dp_i \langle x_i|\exp(-\frac{i\hat{H}\partial t_i}{\hbar})|p_i\rangle\langle p_i|x_{i-1}\rangle \tag{6}$$

We assume ∂t_i is small, substitute the Hamiltonian, and multiply through:

$$\langle x_i|\exp(-\frac{i\hat{H}\partial t_i}{\hbar})|p_i\rangle \approx \langle x_i|(1-\frac{i\hat{H}\partial t_i}{\hbar})|p_i\rangle = \langle x_i|p_i\rangle - \frac{i\partial t_i}{\hbar}(\langle x_i|\frac{\hat{p}^2}{2m}|p_i\rangle + \langle x_i|V(\hat{x})|p_i\rangle)$$

(7)

where the particle indices of the operators are assumed. Hitting the kinetic energy term on the momentum ket and the potential term on the position bra and factoring yields:

$$\begin{aligned}\langle x_i|\exp(-\frac{i\hat{H}\partial t_i}{\hbar})|p_i\rangle &\approx \langle x_i|p_i\rangle\left(1-\frac{i\partial t_i}{\hbar}\left(\frac{p_i^2}{2m}+V(x_i)\right)\right)\\ &\approx \langle x_i|p_i\rangle\exp\left(-\frac{i\partial t_i}{\hbar}\left(\frac{p_i^2}{2m}+V(x_i)\right)\right)\end{aligned} \tag{8}$$

Combining (8) with $\langle x|p\rangle = \frac{1}{\sqrt{h}}\exp(\frac{ixp}{\hbar})$, the integrand of (6) becomes

$$\langle x_i|p_i\rangle\exp\left(-\frac{i\partial t_i}{\hbar}\left(\frac{\hat{p}_i^2}{2m}+V(x_i)\right)\right)\langle p_i|x_{i-1}\rangle = \frac{1}{h}\exp\left(\frac{ip_i(x_i-x_{i-1})}{\hbar}-\frac{i\partial t_i}{\hbar}\left(\frac{p_i^2}{2m}+V(x_i)\right)\right)$$

(9)

We define $\dot{x}_i \equiv \frac{x_i - x_{i-1}}{\partial t_i}$, substitute, and factor:

$$\frac{1}{h}\exp\left[\frac{i\partial t_i}{\hbar}p_i\dot{x}_i - \frac{i\partial t_i}{\hbar}\left(\frac{p_i^2}{2m}+V(x_i)\right)\right] = \frac{1}{h}\exp\left[-\frac{i\partial t_i}{\hbar}\left(\frac{p_i^2}{2m}-p_i\dot{x}_i+V(x_i)\right)\right] \quad (10)$$

Completing the square and factoring gives

$$\begin{aligned}\langle x_i|\exp(-\frac{i\hat{H}\partial t_i}{\hbar})|x_{i-1}\rangle &= \int dp_i\,\frac{1}{h}\exp\left[-\frac{i\partial t_i}{\hbar}\left(\frac{(p_i-m\dot{x}_i)^2}{2m}-\frac{m\dot{x}_i^2}{2}+V(x_i)\right)\right]\\ &= \frac{1}{h}\exp\left[\frac{i\partial t_i}{\hbar}\left(\frac{m\dot{x}_i^2}{2}-V(x_i)\right)\right]\int dp_i\exp\left[-\frac{i\partial t_i}{\hbar}\frac{(p_i-m\dot{x}_i)^2}{2m}\right]\end{aligned} \quad (11)$$

The integral is a Gaussian integral with a complex exponential, which is solvable using a convergence factor. Also, if we define $L(x_i,\dot{x}_i)\equiv\frac{m\dot{x}_i^2}{2}-V(x_i)$, then

$$\langle x_i|\exp(-\frac{i\hat{H}\partial t_i}{\hbar})|x_{i-1}\rangle = \frac{1}{h}\exp\left[\frac{iL(x_i,\dot{x}_i)\partial t_i}{\hbar}\right]\sqrt{\frac{2\pi i m\hbar}{\partial t_i}} = \exp\left[\frac{iL(x_i,\dot{x}_i)\partial t_i}{\hbar}\right]\sqrt{\frac{im}{h\partial t_i}}$$

(12)

Note that we recognize $L(x_i,\dot{x}_i)$ as the Lagrangian. Inserting this expression into (5) yields

$$|\psi_f\rangle = \int\prod_{j=0}^{N}dx_j|x_N\rangle\left(\frac{im}{h\partial t}\right)^{N/2}\exp\left[\frac{i}{\hbar}\sum_{i=1}^{N}L(x_i,\dot{x}_i)\partial t_i\right]\langle x_0|\psi_0\rangle, \quad (13)$$

which is the path integral from $|\psi_0\rangle$ to $|\psi_f\rangle$ using discrete time steps. In some notations [9], a $\mathcal{D}$ is used for the product of differentials. (13) is called a Feynman

path integral. (The above derivation is largely similar to one in Chapter 8 of Townsend. 54)

Note that the sum inside the exponential is a time integral of the Lagrangian on a path described by $\{x_i\}$ (which uniquely determine $\{\dot{x}_i\}$). This sum is the action S along this path:

$$S \equiv \sum_{i=1}^{N} L(x_i, \dot{x}_i)\partial t_i \tag{14}$$

These paths are diagrammatically shown in Figure 1.

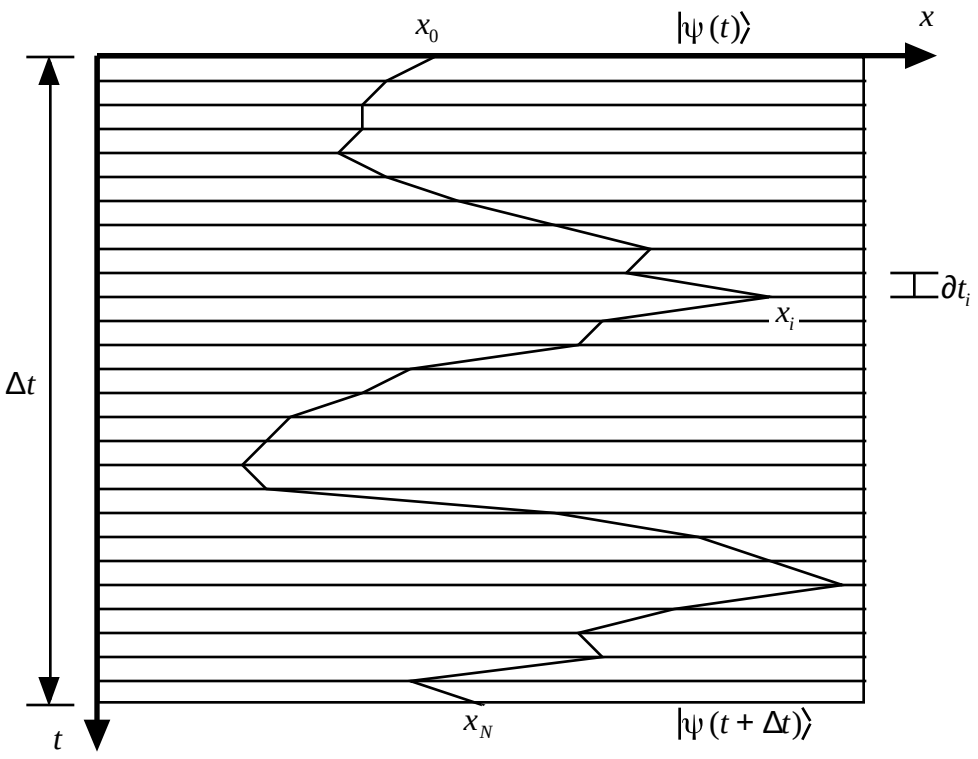


Figure 1. An arbitrary path from x_0 to x_N. Paths like this one link contributions from $|\psi(t)\rangle$ to $|\psi(t+\Delta t)\rangle$ with a phase difference determined by the action on this path.

Note that, at this point, other than the modest requirements used so far, the paths are arbitrary and unrestricted. The particles that follow these paths are called *virtual particles.*

C. The Semiclassical Approximation

We now consider variations $\{\partial x_i\}$ from a special path we label $\{x_{cli}\}$, where $1 \le i < N$. Further definition on the properties of $\{x_{cli}\}$ will be made shortly. We set $x_i = x_{cli} + \partial x_i$ with x_{cli} being independent of x_i, for $1 \le i < N$. From this point forward, let us set $\partial t_i = \partial t = \Delta t / N$. We may apply this substitution to the path integral in (13), but, for the moment, let us focus on the action.

$$S = \sum_{i=2}^{N} \left\{ \frac{m(x_{cli} - x_{cli-1} + \partial x_i - \partial x_{i-1})^2}{2\partial t^2} - V(x_{cl\,i} + \partial x_i) \right\} \partial t + \left\{ \frac{m(x_{cl_1} - x_0 + \partial x_1)^2}{2\partial t^2} - V(x_{cl_1} + \partial x_1) \right\} \partial t \tag{15}$$

We assume $\{\partial x_i\}$ are small and use a Taylor's series expansion of V to organize S in powers of ∂x_i.

$$S = \delta t \sum_{i=2}^{N} \left(\frac{m(x_{cl\,i} - x_{cl\,i-1})^2}{2\delta t^2} - V(x_{cl\,i}) + \frac{m(x_{cl\,i} - x_{cl\,i-1})(\delta x_i - \delta x_{i-1})}{\delta t^2} - \left.\frac{\partial V}{\partial x}\right|_{x_{cli}} \delta x_i + \frac{m(\delta x_i - \delta x_{i-1})^2}{2\delta t^2} - \left.\frac{\partial^2 V}{\partial x^2}\right|_{x_{cl\,i}} \frac{\delta x_i^{\,2}}{2} - \left.\frac{\partial^3 V}{\partial x^3}\right|_{x_{cl\,i}} \frac{\delta x_i^{\,3}}{3!} + \cdots \right) + \delta t \left(\frac{m(x_{cl_1} - x_0)^2}{2\delta t^2} - V(x_{cl_1}) + \frac{m(x_{cl_1} - x_0)\delta x_1}{\delta t^2} - \left.\frac{\partial V}{\partial x}\right|_{x_{cl1}} \delta x_1 + \frac{m\delta x_1^{\,2}}{2\delta t^2} - \left.\frac{\partial^2 V}{\partial x^2}\right|_{x_{cl_1}} \frac{\delta x_1^{\,2}}{2} - \left.\frac{\partial^3 V}{\partial x^3}\right|_{x_{cl_1}} \frac{\delta x_1^{\,3}}{3!} + \cdots \right) \tag{16}$$

Note that the kinetic energy component only contributes to the lowest three orders.

Let us consider with the terms that are first order in δx_i. We now finish the definition of $\{x_{cli}\}$: we define that these values are such that the first order terms in this sum are zero. Since the δx_i are independent of each other, their coefficients must each be zero for this condition to be true. Collecting terms in δx_i, for $1 < i < N$, implies that,

$$-\frac{m(x_{cl\,i+1} - x_{cl\,i})}{\delta t^2} + \frac{m(x_{cl\,i} - x_{cl\,i-1})}{\delta t^2} - \left.\frac{\partial V}{\partial x}\right|_{x_{cl_i}} = 0 \tag{17}$$

Arranging the terms into a more familiar form, we have

$$-\left.\frac{\partial V}{\partial x}\right|_{x_{cl\,i}} = m\,\frac{\dfrac{(x_{cli+1} - x_{cli})}{\delta t} - \dfrac{(x_{cli} - x_{cli-1})}{\delta t}}{\delta t}, \tag{18}$$

and we recognize that this is the time-centered discrete form of $F = ma$. Also

note that the time-discrete velocity expressions are time-centered at half steps relative to the time centering of the position variables. This is consistent with the leap-frog method used to numerically trace *classical* paths. Hence, we recognize that the path described by $\{x_{cli}\}$ is a classical path, justifying its label, cl. Also, it becomes reasonable to name the particles that follow these paths *virtual classical particles*.

Figure 2 depicts a classical path accompanied by its associated variations.

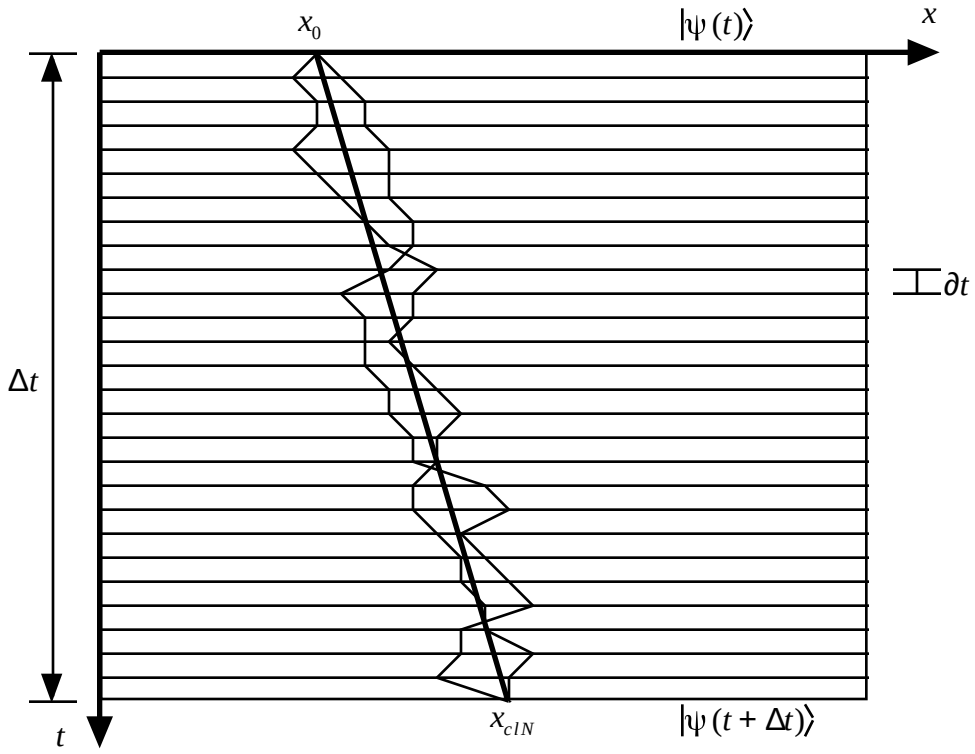


Figure 2. A classical path is shown, accompanied by variations on that path. The virtual classical particles link the quantum wavefunctions.

It is the contributions of a multitude of these classical paths, at a variety

of positions and momenta, that construct the final wavefunction from the initial wavefunction. Also, we make the following distinction: We name ∂t *the classical time step* because it is the time that separates steps of the classical path, but Δt is *the quantum time step* because it is the interval between evaluations of quantum wavefunctions.

D. Initial Position and Final Momentum

We need to consider how to connect the ends of these classical paths to the initial and final wavefunctions. Using the criterion for the term first order in ∂x_1, we have an initial constraint:

$$-\frac{\partial V}{\partial x}\bigg|_{x_{cl\,1}} = m\frac{\dfrac{(x_{cl2}-x_{cl1})}{\partial t}-\dfrac{(x_{cl1}-x_0)}{\partial t}}{\partial t} \tag{19}$$

This links the classical path to the integral over x_0.

Now we consider the final constraint. Let us insert $\mathbf{1} = \int dp_f \left| p_f \right\rangle \left\langle p_f \right|$

before the $\left|x_N\right\rangle$ in (13), resulting in:

$$\left|\psi_f\right\rangle = \int dp_f \prod_{j=1}^{N} dx_j \left| p_f \right\rangle \left(\frac{im}{h\partial t}\right)^{N/2} \frac{1}{\sqrt{h}} \exp\left(-\frac{ix_N p_f}{\hbar}\right) \exp\left(\frac{i}{\hbar} S\right) \left\langle x_0 \middle| \psi_0 \right\rangle \tag{20}$$

Performing the above substitution and requiring that the coefficient of the ∂x_N

be zero implies the following constraint:

$$-\frac{p_f}{\partial t} + \frac{m(x_{cl_N} - x_{cl_{N-1}})}{\partial t^2} - \left.\frac{\partial V}{\partial x}\right|_{x_{cl_N}} = 0 \tag{21}$$

Rearranging gives

$$-\left.\frac{\partial V}{\partial x}\right|_{x_{cl_N}} = \frac{p_f - \frac{m(x_{clN} - x_{clN-1})}{\partial t}}{\partial t} \tag{22}$$

(18) gives N-2 constraints on $\{x_{cli}\}$, and (19) and (22) provide the (N-1)th and Nth constraint, allowing $\{x_{cli}\}$ to be uniquely identified by x_0 and p_f.

Rewriting $|\psi_f\rangle$:

$$|\psi_f\rangle = \int dp_f \int dx_0 \, |p_f\rangle \frac{1}{\sqrt{h}} \exp(-\frac{i x_{clN} p_f}{\hbar}) \exp\left(\frac{i}{\hbar} S_{cl}\right) A \langle x_0 | \psi_0 \rangle \tag{23}$$

where

$$S_{cl} \equiv \partial t \sum_{i=1}^{N} \left(\frac{m(x_{cl_i} - x_{cl_{i-1}})^2}{2\partial t^2} - V(x_{cli})\right), \tag{24}$$

(using $x_{cl0} \equiv x_0$) the zeroth order terms of the action,

$$A = \int \prod_{i=1}^{N} d(\partial x_i) \left(\frac{im}{h\partial t}\right)^{N/2} \exp\left(\frac{i}{\hbar} S_2\right), \tag{25}$$

a N-dimensional integral, and

$$S_2 = \partial t \sum_{i=2}^{N} \begin{pmatrix} \frac{m(\partial x_i - \partial x_{i-1})^2}{2\partial t^2} - \left.\frac{\partial^2 V}{\partial x^2}\right|_{x_{cl_i}} \frac{\partial x_i^{\,2}}{2} \\ - \left.\frac{\partial^3 V}{\partial x^3}\right|_{x_{cli}} \frac{\partial x_i^{\,3}}{3!} + \cdots \end{pmatrix} + \partial t \begin{pmatrix} \frac{m\partial x_1^{\,2}}{2\partial t^2} - \left.\frac{\partial^2 V}{\partial x^2}\right|_{x_{cl_1}} \frac{\partial x_1^{\,2}}{2} \\ - \left.\frac{\partial^3 V}{\partial x^3}\right|_{x_{cl1}} \frac{\partial x_1^{\,3}}{3!} + \cdots \end{pmatrix}, \quad (26)$$

the second order terms and higher of the action. A substitution, $p_f = p_{clf}(p_0)$ (using $p_0 \equiv m\frac{x_{cl1} - x_0}{\partial t}$), can be used to identify these paths using initial conditions only.

E. The Matrix

(This section largely follows Chapter 14 of the Schulman reference [17], with significant points of customization.) Consider S_2. Let us assume that the terms higher than second order in ∂x_i are neglectable. This allows us to write S_2 in the following form:

$$S_2 = \frac{m}{2\partial t}\eta^j M_j^i \eta_i, \quad (27)$$

using the Einstein summation convention, where $\eta \equiv (\partial x_1, \partial x_2, \cdots, \partial x_N)^T$, M is a tridiagonal $N \times N$ matrix,

$$M = u - w = \begin{pmatrix} 2 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2 & -1 & & & \vdots \\ 0 & -1 & 2 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & \ddots & 2 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 1 \end{pmatrix} - \begin{pmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & w_N \end{pmatrix}, \tag{28}$$

and

$$w_i \equiv \frac{\partial t^2}{2m} \frac{\partial^2 V}{\partial x^2}\bigg|_{x_{cl_i}}. \tag{29}$$

For any matrix M, there exists a unitary transformation U so that $M' = UMU^{-1}$ is diagonal. The basis set of M' maps to the eigenvectors of M. In the new basis set, $\eta' = U\eta = (\partial x'_1, \partial x'_2, \cdots, \partial x'_N)$ and M' is diagonal:

$$M' = \begin{pmatrix} m_1 & 0 & \cdots & 0 \\ 0 & m_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & m_N \end{pmatrix} \tag{30}$$

where m_i are the eigenvalues of M (and M'). Therefore S_2 may be rewritten as:

$$S_2 = \frac{m}{2\partial t} \eta'^j M'^i_j \eta'_i = \frac{m}{2\partial t} \sum_{i=1}^{N} \partial x'^2_i m_i \tag{31}$$

This makes A separable:

$$A = \int \prod_{j=1}^{N} d(\partial x'_j) \left(\frac{im}{\hbar \partial t}\right)^{N/2} \exp\left(\frac{i}{\hbar} \frac{m}{2\partial t} \sum_{i=1}^{N} \partial x'^{2}_{i} m_i\right)$$
$$= \left(\frac{im}{\hbar \partial t}\right)^{N/2} \prod_{i=1}^{N} \int d(\partial x'_i) \exp\left(\frac{im}{2\hbar \partial t} m_i \partial x'^{2}_{i}\right) \tag{32}$$

The integrals are Gaussian, so A simplifies:

$$A = \left(\frac{im}{\hbar \partial t}\right)^{N/2} \prod_{i=1}^{N} \sqrt{\frac{2\pi\hbar\partial t}{im\ m_i}} = \left(\frac{im}{\hbar \partial t}\right)^{N/2} \left(\frac{\hbar \partial t}{im}\right)^{N/2} \frac{1}{\sqrt{\prod_{i=1}^{N} m_i}} = \frac{1}{\sqrt{\det(M')}} \tag{33}$$

because the determinant of a diagonal matrix is the product of its elements. But since $\det(M') = \det(UMU^{-1}) = \det(U)\det(M)\det(U^{-1}) = \det(M)$,

$$A = \frac{1}{\sqrt{\det(M)}} \tag{34}$$

(There are issues concerning when this determinant goes to zero, but that will be addressed in the next section.)

Then $|\psi_f\rangle$ becomes:

$$|\psi_f\rangle = \int dp_f \int dx_0 \, |p_f\rangle \frac{1}{\sqrt{h \det(M)}} \exp(-\frac{i x_{clN} p_f}{\hbar}) \exp\left(\frac{i}{\hbar} S_{cl}\right) \langle x_0 | \psi_0 \rangle \tag{35}$$

a two-dimensional integral, with M defined above.

F. The Determinant

(At this point, this discussion substantially diverges from Schulman's [17] and, to the author's knowledge, is not expressed elsewhere.) Let us take a closer look at evaluating the determinant of the above matrix. At first glance, it appears calculating this determinant may be necessary to allocate at least O(N) storage, but an alternative approach was developed to reduce the storage to O(1). This approach was developed to find a convenient form to calculate it numerically, but it also shows the likelihood of it causing the determinant to become singular, which is the results from the "conjugate points" and "caustics" studied at length in other references [10,13,17-19,51].

Let us consider the determinant of an $i \times i$ upper-left minor of M and call it d_i. For $2 < i < N$,

$$d_i = \begin{vmatrix} 2-w_1 & -1 & 0 & \cdots & \cdots & 0 \\ -1 & 2-w_2 & -1 & & & \vdots \\ 0 & -1 & 2-w_3 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & \ddots & 2-w_{i-1} & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 2-w_i \end{vmatrix} \tag{36}$$

Evaluating this determinant by minors gives:

$$d_i = (2-w_i)\begin{vmatrix} 2-w_1 & -1 & 0 & \cdots & 0 \\ -1 & 2-w_2 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & 2-w_{i-2} & -1 \\ 0 & \cdots & 0 & -1 & 2-w_{i-1} \end{vmatrix} - (-1)\begin{vmatrix} 2-w_1 & -1 & 0 & \cdots & 0 \\ -1 & 2-w_2 & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & 2-w_{i-2} & 0 \\ 0 & \cdots & 0 & 0 & -1 \end{vmatrix} \tag{37}$$

But we may recognize that the first determinant is d_{i-1} and the second becomes $(-1)d_{i-2}$.

Therefore,

$$d_i = (2-w_i)d_{i-1} - d_{i-2} \tag{38}$$

There are a few special cases: For d_N, d_1, and d_2:

$$d_N = (1-w_i)d_{N-1} - d_{N-2} \tag{39}$$

$$d_1 = (2-w_1) \tag{40}$$

$$d_2 = (2-w_2)d_1 - 1 \tag{41}$$

Or (38) may be used to calculate d_2 and d_1 if we define

$$d_0 \equiv 1 \tag{42}$$

and

$$d_{-1} \equiv 0 \tag{43}$$

The above expressions provide a complete description, in the form of an iterative method, for evaluating the determinant of M. Algorithmically, this

evaluation can be performed alongside the evaluation of the classical path using (18) with a minimum of storage space, because w_i is only a function of x_{cli}.

Rearranging (38) gives

$$(d_i - 2d_{i-1} + d_{i-2}) + w_i d_{i-1} = 0 \tag{44}$$

We recognize that this is a time-discrete leapfrog-method form of the following ordinary differential equation:

$$\frac{dy(t)}{dt} + w(t)y(t) = 0, \tag{45}$$

which is the simple harmonic oscillator equation with a time-dependent frequency term, where $y(t) = d_t$ and $w(t) = w_{t+1} = \frac{\partial t^2}{2m}\frac{\partial^2 V}{\partial x^2}\Big|_{x_{cl_{t+1}}}$. Interpreting (40) and (41) in this context implies the following initial conditions on y:

$$y(0) \equiv 1 \tag{46}$$

$$\dot{y}(0) \equiv 1 - w_1 \tag{47}$$

The determinant is given by $y(N) - y(N-1)$.

Let us investigate the likelihood of d_N becoming zero. For the sake of argument, let us make w constant. If $w = 0$, then y begins at 1 and increases linearly without bound, resulting in a determinant of 1. If $w < 0$, which corresponds to a defocusing V, then y will increase without bound exponentially, resulting in a determinant greater than 1.

However, if $w > 0$, corresponding to a V that focuses, then y will behave

as a sine wave with a period of:

$$T = \frac{2\pi}{\sqrt{w}} \tag{48}$$

Because the initial conditions are non-zero with a positive slope and w is typically less than 1, $\dot{y}$ will not become zero within one eighth-period. Therefore, if we wish to be sure of never encountering a path whose determinant becomes zero, then

$$N < \frac{\pi}{4\sqrt{w}} = \frac{\pi}{4\partial t}\sqrt{2m\Big/\frac{\partial^2 V}{\partial x^2}}\,, \tag{49}$$

but ∂t is dependent on N, so the requirement becomes

$$\Delta t < \frac{\pi}{4}\sqrt{2m\Big/\frac{\partial^2 V}{\partial x^2}} \tag{50}$$

Here we have a recommended upper bound on Δt, the time between quantum wavefunction evaluations, depending on the physics of the system. This is a worst case scenario, when V has a period of sustained focusing (e.g., in the simple harmonic oscillator). To the author's knowledge, this prediction (50) is not made and utilized elsewhere.

For typical physical parameters, however, other issues, such as changes in the effective V due to the movement of other particles, will require a Δt significantly smaller than required by (50). In practice, the period is long enough (or $\frac{\partial^2 V}{\partial x^2}$ is small enough) so that d_N, at worst, remains within 1% of 1.

G. Summary

We now have a method to time-evolve quantum wavefunctions using classical calculations designed for computation. Here we gather the equations in preparation for implementation. We calculate the following double integral:

$$|\psi(t+\Delta t)\rangle = \int dp_0 \int dx_0 \left| p_{clf} \right\rangle \left(\frac{\partial p_{clf}}{\partial p_0} \right) \frac{1}{\sqrt{h \det(M)}} \exp\left(-\frac{i x_{clN} p_{clf}}{\hbar}\right) \exp\left(\frac{i}{\hbar} S_{cl}\right) \langle x_0 | \psi(t) \rangle \tag{51}$$

A large number of classical paths, each uniquely identified by the dummy variables x_0 and $p_0 \equiv m\frac{x_{cl1} - x_0}{\partial t}$ are traced using:

$$-\left.\frac{\partial V}{\partial x}\right|_{x_{cl\,i}} = m\,\frac{\frac{(x_{cli+1} - x_{cli})}{\partial t} - \frac{(x_{cli} - x_{cli-1})}{\partial t}}{\partial t} \tag{18}$$

over $N = \Delta t/\partial t$ time steps (using $x_{cl0} \equiv x_0$). The action along each path, S_{cl}, is given by:

$$S_{cl} \equiv \partial t \sum_{i=1}^{N} \left(\frac{m(x_{cl\,i} - x_{cl\,i-1})^2}{2\partial t^2} - V(x_{cli}) \right) \tag{24}$$

Simultaneous with the evaluation of each classical path, $\det(M)$ is calculated using an iterative method:

$$d_i = (2 - w_i) d_{i-1} - d_{i-2}, \tag{38}$$

for $1 \leq i < N$, using initial conditions

$$d_0 \equiv 1 \text{ and } d_{-1} \equiv 0 \qquad (42) \text{ \& } (43)$$

where

$$w_i \equiv \frac{\partial t^2}{2m} \frac{\partial^2 V}{\partial x^2}\bigg|_{x_{cl_i}} \qquad (29)$$

The determinant itself is

$$\det(M) = (1 - w_N) d_{N-1} - d_{N-2} \qquad (52)$$

Finally, the final classical momentum, p_{clf}, is given by:

$$-\frac{\partial V}{\partial x}\bigg|_{x_{cl\,N}} = \frac{p_{clf} - \frac{m(x_{clN} - x_{clN-1})}{\partial t}}{\partial t} \qquad (53)$$

This completes the time evolution of $|\psi\rangle$.

III. Implementation

A. The Numbers

We now need to focus on implementing the methods described in the last chapter to a numerical technique appropriate for current computer hardware. This chapter defines and details the organization of these semiclassical calculations to evolve quantum wavefunctions. The following presentation introduces methods and results that are new and have not been located in any previous literature.

The total wavefunction is assumed to be separable into wavefunctions for each particle.

$$|\Psi\rangle = \prod_l |\psi_l\rangle \tag{54}$$

We represent each wavefunction on a set of grid points in space, thus discretizing the wavefunctions. Each $\psi_l(x) \equiv \langle x|\psi_l\rangle$ is a complex number. All

wavefunctions are begun with a complete description of their initial state at $t = 0$. At any time t, the information contained in all the $\psi_l(x)$'s *alone* is used to update the wavefunctions to the next Δt.

(51) contains a prescription for the organizing the classical paths. The obvious solution is to approximate the integral over x_0 with a sum, and assign values of x_0 to the grid points used to represent $\psi_l(x)$. However, what is missing is how to link these paths to the grid point representation of the final wavefunction. Clearly defining this link is very important for the correct evolution of these discretized wavefunctions. We show this link by hitting a $\langle x_f|$ bra on both sides of the equation. (51) becomes

$$\langle x_f|\psi(t+\Delta t)\rangle = \sum_{p_0}\sum_{x_0} \exp(\frac{ix_f p_{cl\,f}}{\hbar})\exp(-\frac{ix_{cl\,N} p_{cl\,f}}{\hbar})\frac{\exp(iS_{cl}/\hbar)}{h\sqrt{\det(M)}}\langle x_0|\psi(t)\rangle \Delta x \Delta p \qquad (55)$$

(We assume the majority of the effects on this value will be due to phase variations between classical paths, expressed in S_{cl}, therefore we assume $\left|\frac{\partial p_{cl\,f}}{\partial p_0}\right|$ varies negligibly from 1.) Each $\psi_l(x_f, t+\Delta t)$ acquires the value of a

double sum. Note that the classical paths can weave, and end, in between grid points and *at the same time* (55) provides a means to link the initial and final wavefunctions on the same set of grid points. This feature is not provided in other theoretical studies of the semiclassical method.

One other issue to examine is the range of momenta. The paths of the original path integral essentially explore all of phase space. The conversion to classical paths allows us to "strategically poll" phase space, but the sampling needs to be just as thorough. We have established that x_0 will range over all grid points, which is the entire space of the calculation, so it seems reasonable to say that p_0 will range over all momenta of the calculation. What is the range of possible momenta of this calculation? The Nyquist theorem states that a maximum frequency can be represented on a series of grid points in time. This theorem has a simple extension to the greatest momenta that can be represented using grid points in space.

$$p_{\max} = h/2\Delta x \tag{56}$$

where Δx is the grid spacing and h is Planck's constant. Since the representation is complex, negative momenta are allowed, so the range of p_0 is $-p_{\max} < p_0 < p_{\max}$. The resolution of the momentum representation of the wavefunction, $\psi_l(p)$, is the same as that of the position representation. Since we justified the spatial resolution using $\psi_l(x)$, it seems reasonable that the resolution of the p_0 distribution should be at least that of $\psi_l(p)$. Although this is not a formal argument, the success of this momentum distribution has been seen empirically.

The general prescription for time evolving the wavefunction is as follows, guided by (55) from right to left.

- Particle Preparation - Begin with a large array of virtual classical particles. Each starts from a grid point, x_0, of $\psi_l(x,t)$, and particles that start from the same grid have a range of initial momenta, $|p_0| < p_{\max}$. With each virtual classical particle, remember the value of the initial wavefunction at the particle's start, $\psi_l(x_0,t)$.
- Particle Pushing - Trace the classical path of each particle using (18), but accumulate its action using (24), and determinant values using (38).
- Particle Depositing / Wavefunction Reconstruction - For each virtual classical particle, at the end of its path, calculate the product of the initial wavefunction, square root of the determinant, and complex exponentials based on the information contained in $\psi_l(x_0,t)$, S_{cl}, x_{clN}, p_{clf}, and $\det(M)$ (using (52)). Then x_f ranges over all the grid points in $\psi_l(x,t+\Delta t)$. The complex number resulting from the classical path is then multiplied by the leftmost complex exponential in (55) and this product is accumulated into $\psi_l(x_f,t+\Delta t)$. Completing this task for all x_f finishes the deposit of that virtual classical particle into the final wavefunction. Completing this task for all such particles reconstructs the entire final wavefunction.

The above procedure assumes that the effective potential on each ψ_l have been established prior to the particle pushing. The details of that calculation depend

on the selected physics of the problem, such as interactions between quantum particles.

There are a few points to note:

- Once all $\psi_l(x, t+\Delta t)$ are complete, the procedure may start anew to calculate $\psi_l(x, t+2\Delta t)$, and so on.
- The amount of data retained between quantum time steps are no more than that of the $\psi_l(x)$.
- Once $\psi_l(x,t)$ (and the effective potential) at a particular time t is established, the calculations preparing, pushing, and depositing the virtual classical particles can be performed in any order. This observation encourages us to use a style of implementation suitable for computers with multiple processors.

B. The Plasma PIC Code

Particle-in-Cell (PIC) implementations have been used with great success in modeling plasmas. Such an implementation assumes a particle-based model of a plasma. In contrast to a fluid-based model, which calculates the result of a finite-difference form of differential equations that assume the plasma behaves as a continuum, the particle model calculates the motions of a multitude of

individual particles. These particles, possessing mass and charge, follow motions due to their mutual electromagnetic fields in a way consistent with classical mechanics (i.e., the extremum of the action, consequently $F = ma$). The simulations successfully show plasma dynamics using only this "first-principles" approach.

The Plasma Physics Group at UCLA has developed efficient and effective methods for using parallel computers to carry out PIC simulations. Their codes, achieving ≈90% parallelism and ≈40% of estimated peak speed, have handled over 2×10^8 particles on 32×10^6 grid points in three-dimensions.[27-33] The methods are robust and portable[34,35] and have run successfully on a wide range of parallel computers (e.g., Cray-90's, T3Ds, T3Es, and IBM SP2s).

What is interesting to note is that much of the success of the plasma PIC code is possible because of how well it manages the simulation of and interactions between a very large collection of particles obeying classical behavior. Knowing the demonstrated success of such techniques in modeling plasmas, J. M. Dawson conceived of the idea to apply the same techniques to managing the classical paths expressed in the semiclassical methods derived for quantum mechanics. A plasma PIC code was converted to a quantum PIC code, but, because of the importance of the structure of the plasma PIC code to a quantum simulation of this type, we describe the plasma PIC code here. In this particular case, the code assumes that the interactions are electrostatic.

The particles in the plasma simulation are distributed in a region of space. Each particle is defined to have a position and velocity in this space. Also, to facilitate the interaction calculations, a regular set of grid points are defined in this space. The particles usually significantly outnumber the grid points, and may reside anywhere in between the grids. The particles, residing in a space of grid-points, is depicted in Figure 3.

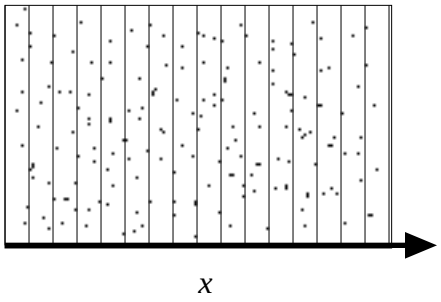


Figure 3. Particles in space overlaid with a grid in one dimension.

Figure 4 shows a simplified flow chart for the plasma PIC code.

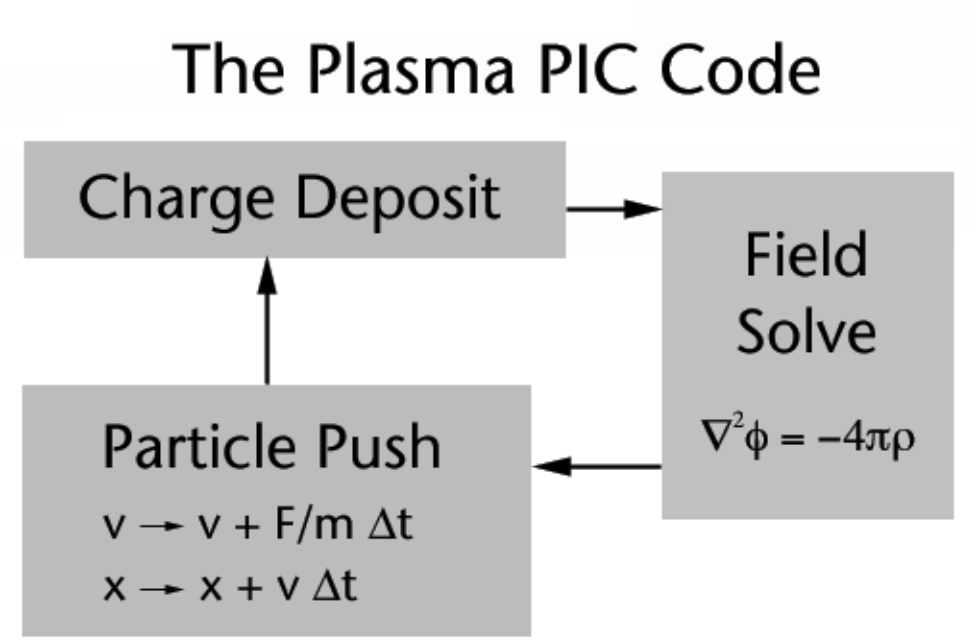


Figure 4. Simplified flow chart for the plasma PIC code.

The code has three major steps:

- Charge Deposit - Given the positions and charges of the particles, the charge deposit routine accumulates the charge contributions due to each particle onto a the grid defined by the grid points. Often, the particles are considered to have a width comparable to the grid spacing, so their charge contributes to more than one grid point. A number of charge sharing techniques are also used. This routine generates a charge density on a grid throughout space.
- Field Solve - Then, the electrostatic field and electrostatic potential are calculated given the charge density from the depositor. It generates this information by performing a Fast Fourier Transform (FFT) of the charge

density, then multiplying this density in Fourier space by a kernel corresponding to the Poisson equation, finally using an inverse FFT to generate the electrostatic field and potential. The speed of this method of solving for fields largely motivates the use of grid points.

- Particle Push - Here, the velocities and positions of the particles are updated using the electrostatic field. The field at a particle's position is interpolated from the electric field at neighboring grid points. Often the leap-frog method is used here for its balance of stability, speed, and accuracy. The routine calculates one leap-frog iteration per particle. The bulk of the CPU time is spent here, due to the sheer number of particles.

Once the particle push is finished, the particles have new positions, which are used in the charge deposit to repeat the process.

Described so far is how the plasma code as a whole works, but it becomes important how to implement this computation on parallel computers. The key is how to organize the work distributed between processors. Here we use a parallel implementation of a technique originally named General Concurrent Particle-in-Cell.[55] Today it is commonly referred to as Particle-In-Cell (PIC), and we use a version of it for parallel computers called Parallel PIC. On a parallel system containing N processors, the space is divided into N regions called cells. Each processor is responsible for the physics inside its cell. This means that every step in the plasma code must be partitioned in this way.

A portion of particles is in each cell, hence the name of the approach. The particles, partitioned by cells, is depicted in Figure 5.

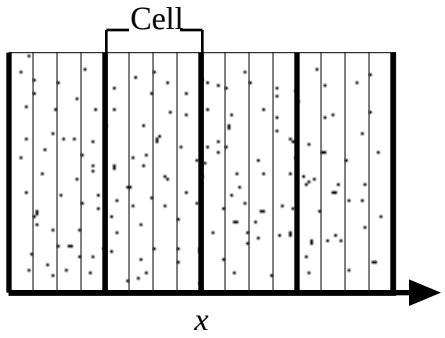


Figure 5. Particles in space partitioned into four cells.

Each processor is responsible for the particles and grid points assigned to it, and communicates with the other processors only when necessary. For example, the charge depositor is easy to organize, since each processor is depositing particle charge only on to its own grids. Only at the end of this step do the processors need to stitch together the charge densities at the edges of their cells.

The field solver is more complicated, as it needs to accomplish FFTs on a grid distributed across multiple processors. In one-dimension, this problem is handled using a custom version [56] of the FFT algorithm designed for this purpose.

Once the electrostatic field information is properly distributed between processors, the particle pusher on each processor updates the particle positions

and velocities as in the nonparallel case, but some of the particles may move out of their original cell. A particle manager must then determine which particles have left its original cell and which processor they should subsequently reside in. It then forwards each particle to the processor corresponding to the cell it just entered.

C. The Quantum PIC Code

We now describe a prescription to create the quantum PIC code from the plasma PIC code. The quantum PIC code has significant organizational differences from its plasma counterpart. Rather than the classical particles containing the primary description of the simulation from time step to time step, it is the quantum wavefunctions, $\psi_l(x,t)$, that contain the most important information. We borrow the grid-point formalism used for the field solve and apply it to the definition of the spatial discretization of the quantum wavefunctions. Therefore, in the code, $\psi_l(x,t)$ is represented using a complex array identical in dimension to the array used for the electric potential.

Before the main time step loop begins, we initialize the code by allocating all needed arrays and loading the wavefunction arrays with the desired initial conditions. The initial conditions are a simple matter of calculating or loading the desired wavefunctions. For example, Listing 1 shows

a loop over partition `k` of wavefunction `l` that loads each wavefunction in the array `wfcn` with Gaussian wavefunctions of standard deviation 4 centered at positions given by real array `initialPosition` with average momenta given by real array `initialMomentum`.

```
 do k=1,nblok
   joff = noff(k) - 2
   do l=1,nspecies
     pkx = twopi*initialMomentum(l)
     do j=1,nxpmx
        wfcn(j,l,k) = exp(-0.25/(4**2)*(j + joff - &
& initialPosition(l))**2) * &
& cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff)))
     enddo
   enddo
 enddo
```

`Listing 1`: A Fortran code example loading the wavefunctions with Gaussians.

where `noff(k)` contains the coordinate of the first grid point of cell `k`, `twopi` is 2π, `nspecies` is the number of quantum wavefunctions, and `nxpmx` is the number of grid points per cell. The structure of the nested loops is consistent with the partitioning method expressed in a work by Decyk.[34]

We show a simplified flow chart for the quantum PIC code in Figure 6 and subsequently describe each major section of the code.

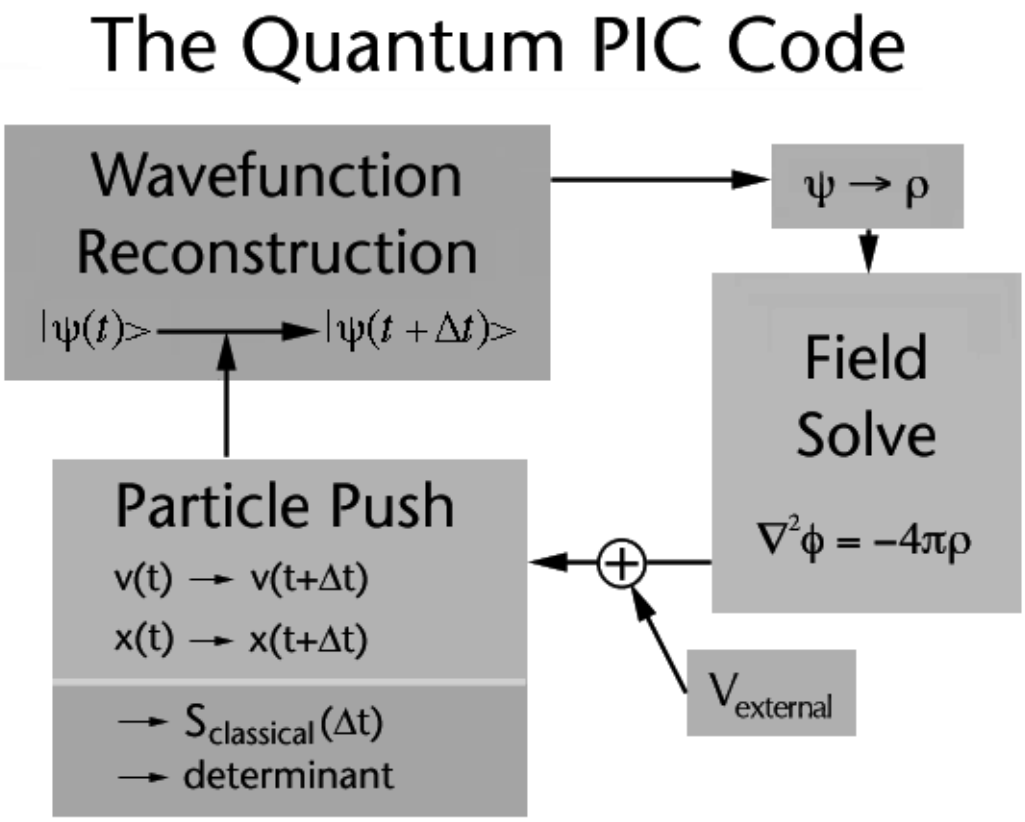


Figure 6. Simplified flow chart for the quantum PIC code

- Charge Deposit

We begin with the charge deposit. The plasma code's original depositor is inappropriate here, but the replacement is much simpler. The charge that quantum particle l encounters is computed by:

$$\rho_l(x) = \sum_{l' \neq l} q_{l'} |\psi_{l'}(x)|^2 \tag{57}$$

where q_l is the charge for particle l (e.g., for electrons, $q_l = -e$). This particle sees the total charge density minus the charge due to itself, which prevents self-interaction. Since ψ at a grid point contributes to charge at the same grid point,

this step is very easy to implement and poses no problems for parallelism. A code example is in Listing 2.

```
      do k = 1, nblok
        do j = 1, nxpmx
          do l=1,nspecies
! initialize charge density to zero
            q(j,l,k) = 0.0
          enddo
        enddo
      enddo
! deposit charge   using qme |wfcn|^2 leaving out self-
interaction
      do 1190 k = 1, nblok
! only where we need to
      do 1180 j = 1, nxp(k)
      do l=1,nspecies
      qiw = qme*(real(wfcn(j+1,l,k))**2 +
aimag(wfcn(j+1,l,k))**2)
        do lt=1,nspecies
          if (l.ne.lt) then
            q(j+1,lt,k) = q(j+1,lt,k) + qiw
          end if
        enddo
      enddo
 1180 continue
 1190 continue
```

Listing 2. Charge depositor for wavefunctions.

q is the charge density array, partitioned across processors, to be used in the field solve.

- Field Solve

Next, we use the same field solve routines, unchanged from the plasma PIC code, to provide the electric field and the electric potential. We will need the field for the classical path calculation and the potential for the classical action

calculation. Note that, for N quantum wavefunctions, N different charge densities will be produced, so N field solves are necessary in one quantum time step. A natural solution is to set up a loop over quantum particle inside the time-step loop.

- External Potential

In addition, we have the option of adding an external potential to the field and potential arrays. This feature is useful for textbook examples, such as the simple harmonic oscillator. In one dimension, it is a simple task of accumulating the calculated potential into the electric potential array and its negative derivative into the electric field array. This approach allows for time-dependent external fields as well. These calculations are local, providing for easy parallelization.

With the wavefunction and field arrays ready, we now proceed to the procedures that calculate (55). To store information about the virtual classical particles, we borrow the plasma code's original particle array and extend it. Formerly, each entry of this array had information only about position and velocity. In the quantum PIC code, we must add allocation for the action (1 real), the determinant (2 reals for d_i and d_{i-1}), and the value of the original wavefunction (1 complex). In the one-dimensional code, this increases the

number of real numbers allocated per classical particle from 2 to 7.

- Particle Preparation

The virtual classical particle array is initialized using a new particle preparation routine. The initial conditions of these virtual classical particles are determined by the indices of the double sum expressed in (55). The particles' initial positions are evenly distributed across all grid points of the wavefunction. Particles beginning at the same grid point have a regular distribution of initial momenta, bounded by (56). For all particles: the action attribute of the particle is initialized to zero; the determinant information are initialized according to

$$d_0 \equiv 1 \text{ and } d_{-1} \equiv 0; \qquad (42) \text{ \& } (43)$$

and the initial wavefunction information is set to the wavefunction evaluated at the grid point from which the virtual classical particle begins. Note that this particle initialization uses operations are entirely local (copying wavefunction data and other initialization) or are very regular (initial positions and velocities), making this routine simple to parallelize. (This is in contrast to particle initialization in the plasma code, which must generate a psuedo-random distribution while guaranteeing no correlations of particle data between processors. The typical solution is for all processors to generate identical distributions of the entire plasma and disposing the particles outside of their assigned cells, providing no parallelism.)

- Particle Push

With the virtual classical particles initialized, the code is ready to push them through space. This involves a modified version of the plasma code's particle pusher. Since the classical path specified by (18) is the same provided by the leap-frog method driven by the force described by the electric field array, that portion of the routine is unchanged. The pieces we must add are the action accumulation and the determinant evaluation. The accumulation of the action is given by (24), which uses the electric potential array and the velocity, time-centered at half steps, given by the leap-frog method. The evolution of the determinant is given by (38).

Listing 3 gives a code sample that advances the particle position and velocity according to the leapfrog method (18) and the classical action (24), assuming that `ax` and `px` are loaded with the electric field and the potential at the particle's previous position.

```
! new velocity
      dx = part(2,j,k) + qtm*ax

! action accumulate
      part(3,j,k) = part(3,j,k) + .5*dt*((dx)**2) - qtm*px
! new velocity
      part(2,j,k) = dx
! new position
      part(1,j,k) = part(1,j,k) + dx*dt
```

Listing 3. A code listing for one virtual classical particle push.

where `dt` is the classical time step, `qtm` is the product of the particle's charge to

mass ratio and dt, and part is the particle array. Then we have the determinant calculation in Listing 4.

```
! push det's, using (∂t)^2 V'' / m
! = (dt/vscale)^2 (qtm*(vscale**2)/dt)*pt'' / m
! = pt''*qtm*dt
      px = pt(nn+1,l,k) + pt(nn-1,l,k) - 2.0*pt(nn,l,k)
      ax = part(7,j,k)
      part(7,j,k) = part(6,j,k)
      part(6,j,k) = (2.0 - px*qtm*dt) * part(7,j,k) - ax
```

Listing 4. Determinant evolution code sample.

where nn is the nearest neighboring grid point to the current position of the particle. (vscale will be defined in a later section.) px acquires the value of the second derivative of the potential, which is used to calculate the latest determinant value in part(6,j,k). Here it is possible to check if the determinant is going become singular.

- Particle Manager

Since the particles, after one push, could venture outside their initial cell, the particle manager of the plasma code must be invoked here. Other than the simple change to allow for more data per particle, it is identical to the plasma PIC code's particle manager. Parallelization issues for both the pusher and the manager are identical to that of the plasma code.

One important conceptual difference we must emphasize here, however, is that the particle push / particle manager pair is evaluated many times. One

push corresponds to one classical time step ∂t, which subdivides the quantum time step Δt, as related by (4). The ratio of the quantum time step to the classical time step is how many times the particle push and particle manager must be evaluated.

- Wavefunction Reconstruction

Finally, we focus on the wavefunction reconstruction routine. This routine computes "the deposit of complex charge" due to the virtual classical particles, which has a vague analogy to the plasma code's charge depositor, hence an alternative name of "wavefunction depositor". This routine is new and unique to the quantum PIC code.

One way of looking at the implementation of this code is by looking at (55) the following way:

$$\langle x_f | \psi(t + \Delta t) \rangle = \sum_{p_0} \sum_{x_0} \exp\left(\frac{i x_f p_{cl\,f}}{\hbar}\right) f(x_0, p_0) \tag{58}$$

where

$$f(x_0, p_0) \equiv \exp\left(-\frac{i x_{cl\,N} p_{cl\,f}}{\hbar}\right) \frac{\exp\left(iS_{cl}/\hbar\right)}{h\sqrt{\det(M)}} \langle x_0 | \psi(t) \rangle \Delta x \Delta p \tag{59}$$

p_{clf} and f are computed from the particle data calculated by the particle pusher using (24), (52), (53), and (55). The focus of the problem is on x_f. Instead of

summing in the order suggested strictly by (58), where one x_f is considered, and all classical particles are summed, instead consider one virtual classical particle, and distribute its contributions to the final wavefunction as a function of x_f.

Besides computing f only once per virtual classical particle, this approach allows the following scheme for parallelization. Each processor allocates a temporary complex array large enough for a complete description of the wavefunction for all cells. Each processor then accumulates the contributions from its assigned virtual classical particles into its array according to (58). These steps so far require no interprocessor communication.

With the wavefunction buffers complete, these arrays then need to be summed between processors. The temporary array each processor is holding is partitioned into sections designated for other processors. Each processor sends to every other processor the data assigned to them, and receives from every other processor the data it is supposed to accumulate to form its section of the final wavefunction. After the final sum is finished, the complete final wavefunction is formed, correctly partitioned between each processor.

Calculating N complex exponentials for N grid points according to (58) can in general be time-consuming. However, the current version of the code uses a code optimization that completes the same task using only two complex exponential calculations instead of N, resulting in an order of magnitude speed-

up for the overall subroutine. We know that the values of x_f are regularly spaced according to

$$x_f = x_{f0} + j\Delta x \tag{60}$$

where Δx is the grid spacing and j is a nonnegative integer. First, compute the two complex values

$$y_0 \equiv \exp(\frac{ix_{f0}p_{cl\,f}}{\hbar}) f(x_0, p_0) \tag{61}$$

$$m \equiv \exp(\frac{i\Delta x p_{cl\,f}}{\hbar}) \tag{62}$$

which cost two complex exponentials and one complex multiply to produce. The contribution to the wavefunction at the lowest value of x_f is simply y_0. For the next value of x_f, multiply m by y_0. Then for the next, multiply by m again, and so on for all grid points.

Implementing this technique in code is shown in Listing 5.

```
      ctemp = wf * &
&  cmplx(cos(phase + pdh * ( nnoff + 1 ) ), &
&        sin(phase + pdh * ( nnoff + 1 ) ) )
      cincr = cmplx( cos(pdh), sin(pdh) )

      do jw=1,nxpmx

        wtemp(jw,kw,k) = wtemp(jw,kw,k) + ctemp

        ctemp = ctemp * cincr
      end do
```

Listing 5. Inner loop for rapid calculation of contributions to all grid points.

`pdh` is the final momentum divided by $\hbar$, `nnoff+1` is the position of the the

first grid point in this cell, `wf` is the value of $f(x_0, p_0)$ given by (59), and `wtemp` is the temporary wavefunction array. This listing assumes that the simulation is organized such that $\Delta x = 1$.

This procedure completes the contribution due to one virtual classical particle, and we repeat the procedure for all particles. After all the contributions are summed, the data in the virtual classical particle array may be discarded, and its allocation may be reused.

- Renormalization and Diagnostics

To preserve the norm of the wavefunction against numerical error, it is a good idea to renormalize the wavefunction data at this time, maintaining

$$\langle \psi_l | \psi_l \rangle = 1, \text{ for all } l \tag{63}$$

With all wavefunctions updated to the new time step, the wavefunction data for this time step can be saved to a file, and a variety of diagnostics can be computed. Parallelization issues involve properly summing values across processors since the wavefunction being diagnosed or renormalized is distributed. The entire procedure repeats to continue the evolution of the $|\psi_l\rangle$.

D. Boundary Conditions

An important issue in the simulation is how to correctly contain the

quantum wavefunctions if they approach the edges of the simulation. The most typical confinement appropriate for quantum mechanics is the infinite square well potential, which implies the boundary condition constraint

$$\psi_l(x) = 0 \text{ for } x = 0 \text{ and } x = L \tag{64}$$

where L is the size of the box.

However, to make this constraint consistent with these semiclassical methods is an involved question. How should the wavefunction *and* the virtual classical particles behave to be consistent with this constraint? A variety of possibilities exist. For the wavefunction, additional guard cells can be added, the wavefunction could be extrapolated beyond the boundaries using functions that preserved continuity and continuity in the higher-order derivatives of the wavefunction, or the wavefunction could be zeroed beyond the boundary. In addition, the particles could be made to reflect (reverse momentum) or not, and their phase (due to the action of their paths) may or may not be adjusted upon hitting the boundary. Finally, the boundary itself could be redefined at fractions of a grid spacing. Of course, there are always combinations of the above.

The answer used in the code was found empirically using eigenstates of the infinite square well as test cases, to be described in Section D of the following chapter. We allow two guard grid points for each boundary, meaning the boundaries are set at grid points at least two grids away from

edges of the simulation. The wavefunction is set to zero at the boundary and outside the well. The virtual classical particles are reflected, their phase receives no additional adjustment, but their reflection point is one-half grid *beyond* the wall, diagramed in Figure 7.

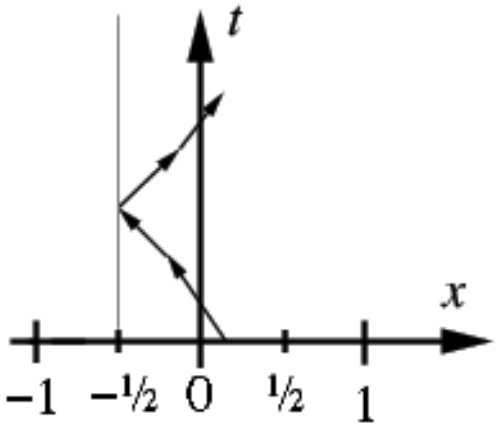


Figure 7. A virtual classical particle reflecting at one-half grid point behind the $\psi_I(x = 0) = 0$ boundary of the well.

The precise theoretical reasons for this phenomena has not yet been determined, but the source of empirical support for this conclusion will be given in the following chapter.

Implementing these adjustments is not difficult. The code to zero the wavefunction is a simple matter to insert after the virtual particle deposit is finished in the wavefunction reconstruction routine. The virtual classical particle reflection is handled in the particle pusher after the leapfrog method step and before the determinant calculation. The code checks if the new particle position is beyond the reflection point. If it is, it resets the new position within

the boundaries and reverses the velocity.

E. Simulation Parameters

We have emphasized on describing the structure of the code in the preceding sections, but, in order to make the code a practical tool, a number of parameters must be set. The choice of parameters reconcile the properties of the algorithms with the practical realities of the numerics. This section is meant to describe the reasons for the particular choices made in the code, providing guidelines for future adjustments or extrapolations.

Since this quantum code is derived from a plasma code, it borrows many features of the plasma code, sufficiently described in other references. [27-33,57] The partitioning issues regarding the organization of grid points between processors are identical, and the techniques used to create "portable parallel" code [34] are carried to any new code unique to the quantum simulation. This prescription allows the code to compile on a variety of parallel computing platforms. It is the recommendation of this author that future users should extend this consistency to any new additions to the code.

Other aspects of the simulation that carry over include the numerical values of the parameters involved. For example, the units of the code are a system designed to make the simulation values easier to handle numerically,

but based on cgs units. For example, the mass of the electron m_e and the grid spacing Δx are fixed at 1. The electron charge e is a parameter on the order of unity, and the time step is a parameter less than one, low enough to prevent the classical particles from stepping over too many grid points in one step and high enough not to expend excessive amounts of CPU time.

Extending on this scheme to quantum mechanics, parameters regarding Planck's constant h, the ratio of the quantum time step to the classical time step $\Delta t/\partial t$, and the maximum momentum $p_{\max}$ must be considered in combination with the size of the discretization of the wavefunction. Given, from the plasma code, that Δx and m are 1, (56) implies that h must be twice the value of $v_{\max}$, the maximum velocity of the virtual classical particles.

h, represented in the code with `planck`, must be chosen to be large enough for the grid spacing of the wavefunction to provide sufficient resolution to represent wavefunctions of interest to us, yet it should not be so large as to waste inordinate amounts of CPU time. 64 was found to be a sufficient value for `planck`, although we suspect 32 could function as well.

`planck = 64` implies $v_{\max} = 32$, however there is more than one way to implement this $v_{\max}$. The method chosen here, primarily for diagnostic purposes, allows flexibility through a number of adjustable parameters. In the plasma code, `vts` is interpreted as the thermal velocity of the plasma and is set to no higher than 1 so that, in combination with `dt`, the classical particles do not

traverse too many grid points too quickly.

We wish to preserve this condition and ease debugging; thus, we leave the form of the particle pusher code unchanged. Therefore, the output of the particle pusher must be reinterpreted with a rescaling of its units. A parameter, called `vscale` in the code, was created to describe the ratio of time scales outside the particle pusher to the time scale inside. This allows the velocity values seen inside the pusher to actually correspond to considerably higher velocities desired outside the routine. The velocities inside must be multiplied by `vscale` before being used with values in the rest of the code. Since action is also proportional to the inverse of the unit of time, action determined by the pusher must also be multiplied by `vscale`. Therefore `vscale*vts` is interpreted as $v_{\max}$. Since `vts` is 1 and `planck` is 64, `vscale` becomes 32 for to satisfy all of the above conditions.

But, not only does this describe the ratio of interpreted velocities inside and outside the pusher, this scenario sets the ratio of the quantum and classical time scales in the code. `tcptq` (short for "number of Timesteps Classical Per Timesteps Quantum"), which represents $\Delta t/\partial t$, sets how many times the particle pusher is called per quantum time step. Since each `dt` inside the pusher is really ∂t outside the routine, `tcptq` must also be equal to `vscale`. It has also been found empirically that `vscale = tcptq = 32` provides a simulation consistent with physics, given `planck = 64` and `vts = 1`.

We ask that the reader bears in mind that this structure is in the code not simply to be confusing. The extra parameters exist because they provided a means to test ideas and alternative schemes until one that worked correctly was found. Further tests on this aspect of the method may become important in the future. However, for those who simply wish to adjust the code for other purposes, we recommend that the user only adjusts `tcptq` and leave rest of the code to set `vts = 1`, `planck = 2 * vscale`, and `vscale = tcptq`.

The last note of empirical knowledge regarding the parameters of the code regards the number of virtual classical particles per quantum particle. The quantum code reinterprets the `nspecies` parameter, which meant the number of different plasma species in the plasma code, instead as the number of different quantum particles. This reinterpretation allows us to use many of the existing mechanisms in the plasma code for organizing particles by species to organize them by quantum particle instead. So, formerly the number of particles per species, `npx` is the parameter used to describe the number of virtual classical particles per quantum particle.

Since these virtual classical particles must start from individual grid points of the initial wavefunction, for a complete and regular sampling, it seems reasonable to say that `npx` should be proportional to the number of grid points, `nx`. So the question becomes what is the number of particles per grid point, `npx/nx`? Since the Fourier transform of a function on `nx` grid points also has a

resolution consisting of nx grid points, one plausible answer is nx. So npx = nx*nx. This hypothesis is borne out by empirical tests: setting npx to be at least this value provides consistent physics, while setting npx below this value causes the wavefunction to shred itself into noise in a few time steps.

However, this is not to say that other solutions are impossible. This simple method of sampling is one that blankets phase space with a density of classical paths sufficient to provide correct results. But it is the belief of this author that there exist solutions that are more clever, some of which will be suggested in Chapter VII - Future Work.

F. Alternative implementations

In the course of developing the theory and the code for this project, a variety of other schemes for almost all aspects of the calculation were also conceived. Those that were attempted are described in Appendix A, some of which may be useful or more appropriate for applications other than those shown in this dissertation. Those that have been speculated upon are described in Chapter VII - Future Work.

IV. Validation

A. Output

After each quantum time step, the quantum PIC code saves data into a variety of files. It runs a series of diagnostics on the wavefunctions, measuring potential energy, kinetic energy, total energy, average position, $\langle \psi_l | \psi_l \rangle$, average momentum, electrostatic energy, and the range of the determinant. In addition, it saves all quantum wavefunctions at all time steps. Besides making it possible to restart the simulation from any point, this quantum data file enables the user to examine the entire time sequence of the simulation for any purpose.

In the early stages of development of this code, these data sets were studied to test the correctness of the simulation. A feedback process was developed to thoroughly test the code against solutions to typical quantum-mechanical problems. Specific well-known phenomena unique to quantum

mechanics were used to probe for possible problems in specific parts of the code. This process was an effort to be sure that the code was as faithful as possible to the physics. The test cases used and their results are described in this chapter. These processes are demonstrated here to provide a guide for future work with this code or future extensions on or extrapolations of this code.

B. Free-Space Gaussian

The first test case studied is the evolution of a single Gaussian in free space. This calculation is a rigorous test of the code because it precisely tests some of the most basic behavior found in quantum mechanics. The initial conditions inserted into the code represented a Gaussian of known standard deviation σ, initial position x_0, and initial momentum p_0 of the form

$$\psi(x,t=0) = \frac{1}{\sqrt{\sigma\sqrt{2\pi}}} \exp\left[-\frac{(x-x_0)^2}{4\sigma^2}\right] \exp\left[i\frac{p_0 x}{\hbar}\right] \tag{65}$$

The space of the code was sufficient in size to allow numerous significant properties of the wavefunction's evolution to be measured before it began to interact with the edges of the simulation in any measurable way. The Gaussian was centered in the space of the simulation, the external potential routine was shut off, and the initial momentum and the charge were set to zero.

For any observable $\hat{A}$, the expectation value of $\hat{A}$ of a wavefunction ψ is calculated using

$$\left\langle \hat{A} \right\rangle = \int \psi^*(x) \hat{A} \psi(x) dx \tag{66}$$

The energy of the wavefunction is measured using the expectation value of the Hamiltonian $\hat{H}$

$$\left\langle \hat{H} \right\rangle = \int \psi^*(x) \left(-\frac{\hbar}{2m} \frac{\partial^2}{\partial x^2} + V(x) \right) \psi(x) dx \tag{67}$$

and the standard deviation using

$$\left\langle (\Delta x)^2 \right\rangle = \left\langle \hat{x}^2 \right\rangle - \left\langle \hat{x} \right\rangle^2 \tag{68}$$

The standard deviation in space of a wavefunction which begins according to (65) will increase as a function of time according to

$$\left\langle (\Delta x)^2 \right\rangle = \sigma^2 \left(1 + \frac{\hbar^2 t^2}{4m^2 \sigma^4} \right) \tag{69}$$

while its total energy remains constant. Frames from the successful modeling of the evolution of this wavefunction are shown in Figure 8.

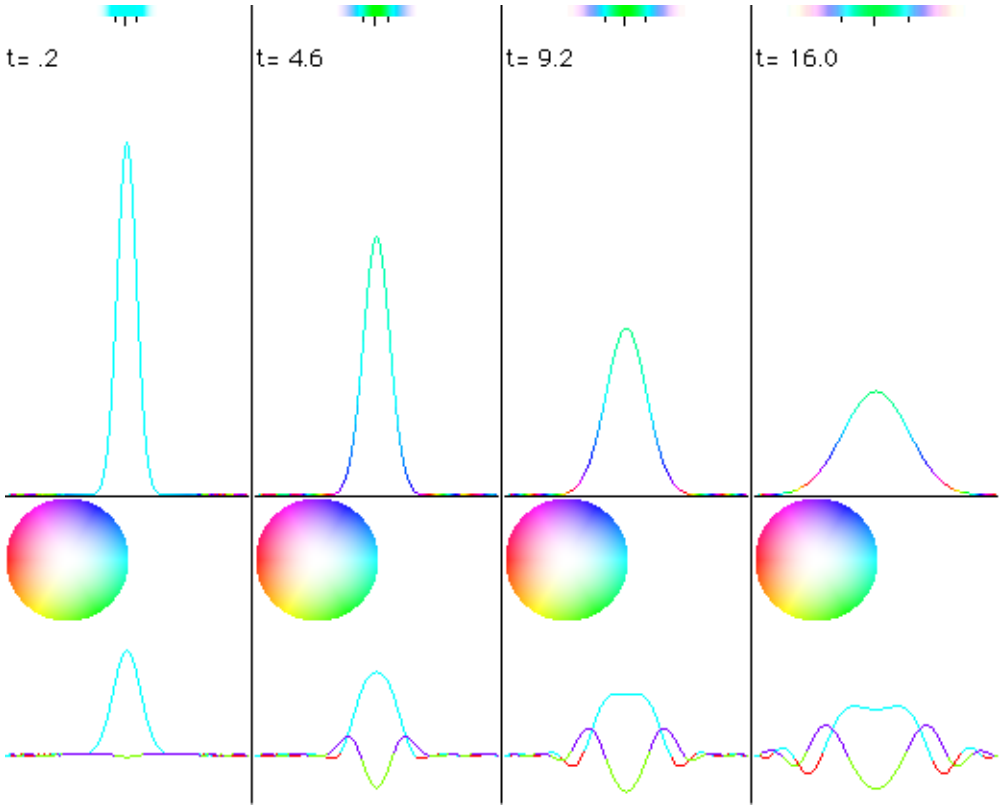


Figure 8. Four frames of the evolution of a stationary Gaussian in free space.

The representation of the wavefunction in Figure 8 is the following. The horizontal axis of all graphs is space. Color indicates the phase of the wavefunction, where cyan is positive real, purple is positive imaginary, red is negative real, and so on around the color wheel. The legend for the phase-color mapping is shown using the color wheel in the frames, assuming a set of real and imaginary axes superimposed on the wheel. This mapping will be used for all further plots of phase. The top bar shows the wavefunction's phase as color and probability density, $|\psi(x)|^2$, as the strength of that color. The middle tick mark is the average position of the wavefunction $\langle \hat{x} \rangle$, and the two

other tick marks are $\langle \hat{x} \rangle - \Delta x$ and $\langle \hat{x} \rangle + \Delta x$, as defined by (66) and (68). The vertical axis of the middle graph is $|\psi(x)|^2$ while color is used for phase, and the lowest graph shows the real and imaginary parts of $\psi(x)$ plotted simultaneously.

The earliest versions of the quantum PIC code presented significant problems. After locating a working range of algorithmic configurations and parameters and debugging the parallel aspects of the new code, the largest of the remaining problems was that the energy of the wavefunction decreased on the order of 1% per time step. The first clues towards the cause of this energy loss was through a careful analysis of the data, including a translation of the wavefunction data into audible sound. A code was developed to translate a data set of floating-point numbers into a format that computer hardware could transform into current impulses delivered to a pair of speakers. At one time step, the real part of the wavefunction was used to supply sound to the left channel, and the imaginary part for the right channel. The speakers play the data in a loop fast enough for the frequency range of the wavefunction to be heard in the audible frequency range. By playing the data at successive time steps, the change in the data, as the wavefunction evolves, can be heard.

What was observed in the data using this technique was that the frequency distribution of the wavefunction was changing from the beginning

of the simulation to the end. In particular it was noted that the higher frequencies, which correspond to the higher momentum components of the wavefunction, were being attenuated as the wavefunction evolved. If the higher momentum components of the wavefunction were decreasing in strength relative to the lower momentum components, that could be enough to explain the energy loss.

Assuming that this higher momentum attenuation was how the energy was being lost guided us to focus on particular parts of the code. This clue led us to the technique used in the wavefunction reconstruction routine to "deposit" the virtual classical particles' contributions onto the grid. At the time, the technique used was a deposit local to the virtual classical particle's final position, very similar to the charge deposit of the plasma code. By analyzing the effect of the deposit as a convolution of the "ideal" wavefunction with a weight function that describes the deposit technique, it was found that the consequence of using this deposit technique could explain both the rate of the energy loss and the high-frequency attenuation heard earlier.

A number of techniques (described in greater detail in Appendix B) were used to attempt to decrease that energy loss, but the solution found to preserve the energy best (that is, with variations that are indistinguishable from round-off error) primarily involved substantial changes in the wavefunction reconstruction routine. Chapters II and III presents the solution that worked

best. This solution produces a wavefunction whose measured width matches (69) as a function of time with variations indistinguishable from round-off error due to the single-precision floating-point variables used. In addition, when the initial momentum is nonzero, the simulation shows a constant translation of the Gaussian between frames consistent with theory.

C. Simple Harmonic Oscillator

The next problem type investigated is the simple harmonic oscillator (SHO) problem. This calculation is among the simplest that requires the quantum particle to interact with its environment and has a behavior that is very well known and easy to recognize. The implementation requires the same conditions and as the free space Gaussian with the addition of an external potential of the form

$$V_{SHO}(x) = \frac{1}{2} m\omega^2 x^2 \tag{70}$$

In the code this potential is implemented as shown in Listing 6. Note that both the potential and its negative derivative must be introduced into the electric potential and field arrays. `omegasq` represents ω^2.

```
real :: omegasq
parameter(omegasq = (1.0/8.0)**2)
```

. . .

```
      do k=1,nblok
        joff = noff(k) - 2
        do j=1,nxpmx

!            ! Simple Harmonic Oscillator
            xt = j + joff - nx/2
            pt(j,l,k) = pt(j,l,k) + &
     & adjustment*0.5*omegasq*(xt**2)
            fx(j,l,k) = fx(j,l,k) - &
     & adjustment*omegasq*xt

        enddo
      enddo
```

Listing 6. Code that introduces a simple harmonic oscillator potential into the electric potential and force arrays.

As demonstrated in Listing 6, the effect of the external potential must be introduced into both arrays consistently. The `adjustment` multiplier is needed because of the unusual units `pt` and `fx` have due to their history as part of a plasma code. `xt` provides the coordinate for the potential while accounting for the partitioning due to the PIC techniques.

The initial conditions for the first test was an arbitrary Gaussian to see if `omegasq` was compatible with the simulation. Once an appropriate value of `omegasq` was selected, a Gaussian corresponding to the ground state of the SHO was used. Frames of this simulation are shown in Figure 9.

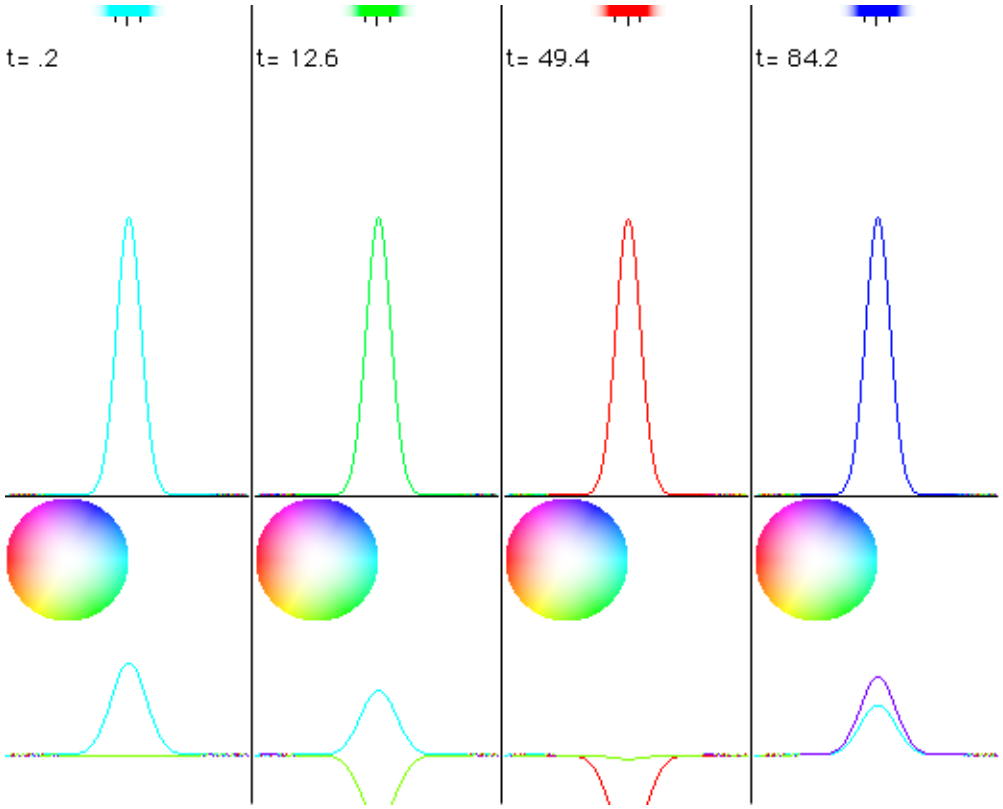


Figure 9. Four frames of the evolution of the ground state of the simple harmonic oscillator.

We were expecting that the simulation would be consistent with the analytical behavior of the ground state, $|n=0\rangle$. In particular, the state should remain as it is with the exception of an evolution in its overall phase. As indicated in Figure 9, the quantum PIC code gave the correct results, and easily maintained the eigenfunctions for hundreds of time steps.

Next, we attempted other SHO eigenstates with higher energy. In particular, we supplied the code eigenstates with quantum numbers n=1, n=5, and n=7. Examples of their structure are shown in Figure 10.

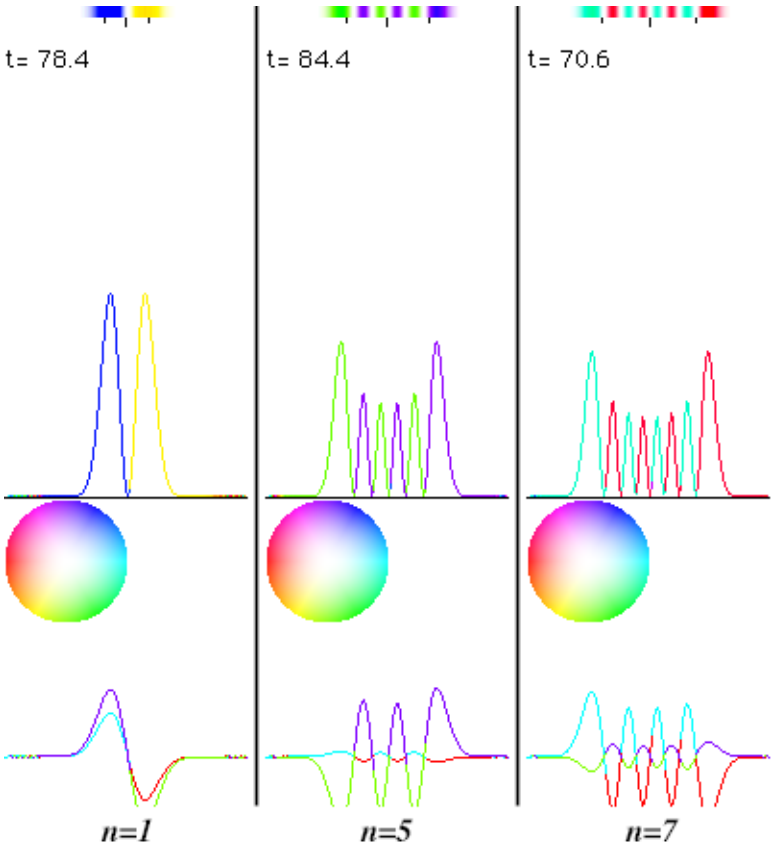


Figure 10. Frames from simulations of the $|n=1\rangle$, $|n=5\rangle$, and $|n=7\rangle$ eigenstates of the simple harmonic oscillator.

What was also seen in the evolution of these eigenstates was that the rate of the evolution of their phase was distinct from each other and in a manner consistent with quantum mechanics, in particular, according to their energy eigenvalues.[54] This is a property that is possible to exploit using a correlation calculation,

$$c(\tau) \equiv \int \langle \psi(t+\tau) | \psi(t) \rangle \, dt \tag{71}$$

or, in the position representation,

$$c(\tau) = \iint \psi^*(x, t+\tau)\psi(x,t)\, dxdt \tag{72}$$

The Fourier transform of the correlation c should show the energy spectra of the system, presenting peaks that correspond to the energy eigenvalues of the eigenstates in the system. Inserting the data from the SHO eigenstate tests each showed one solitary peak, and each peak's frequency corresponding to the energy of the state, as we were expecting.

The next question is: can the code handle multiple eigenstates at once and preserve them independently of each other? To answer this question, a superposition of these eigenstates was used for the initial conditions, evolving as seen in Figure 11.

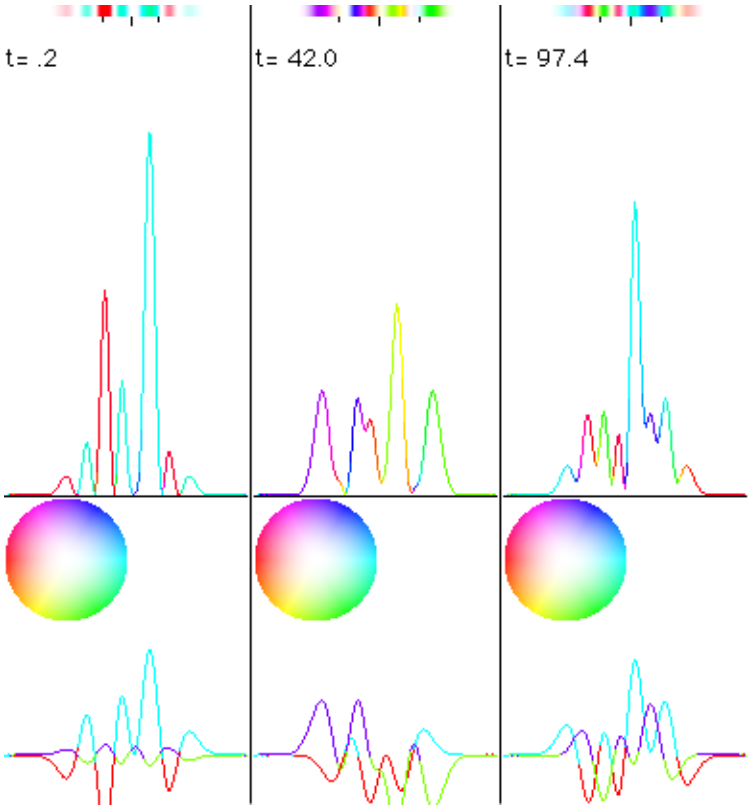


Figure 11. Three frames from the evolution of an arbitrarily chosen superposition of the $|n=0\rangle$, $|n=1\rangle$, $|n=5\rangle$, and $|n=7\rangle$ eigenstates of the simple harmonic oscillator.

The resulting energy spectrum is shown in Figure 12.

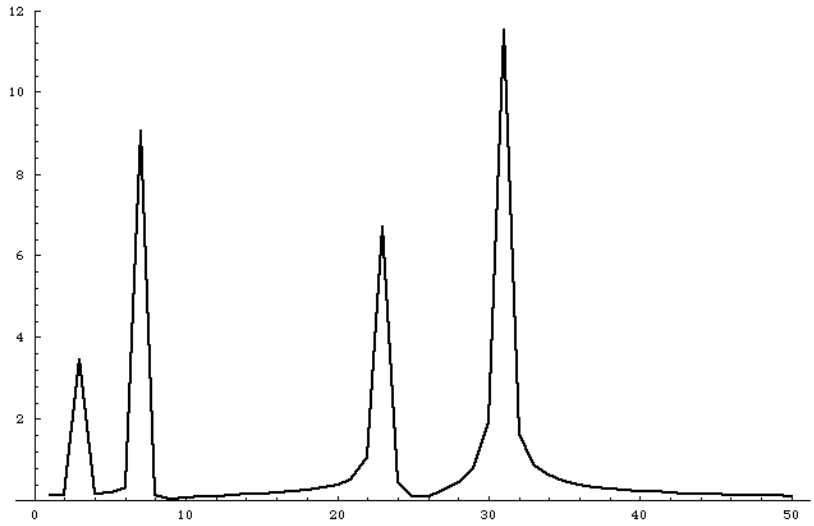


Figure 12. Energy spectrum of the simulation shown in Figure 11.

The energy spectrum clearly shows four peaks, at relative energies that one would expect, based on the theory, for the superposition the code was given.

The reader may recall that this simulation is supported only by a network of classically calculated paths, whose contributions are regularly recoalesced using a process that relies heavily on cancellations due to phase. Before running these simulations, it was conceivable that superpositions might not be successfully maintained because of noise, errors, or crossover between modes of the system. Instead, these results show that the quantum PIC code can maintain a simulation of an arbitrary superposition of eigenstates, and with enough fidelity for the energy spectrum of the system to be extracted from a simulation of sufficient length.

D. Infinite Square Well

The next system of interest is the infinite square well. This system is important because of its primary features: its well-defined and simple boundary conditions:

$$\psi_I(x) = 0 \text{ for } x \leq 0 \text{ and } x \geq L \tag{64}$$

while the wavefunction is unconstrained between these boundaries. This type of constraint is one of the most conceptually convenient ways to precisely define a method to "contain" a collection of particles.

However, the methods to achieve this particular containment and demonstrate this achievement in this quantum PIC code are not immediately clear. A discussion of the possible combinations to attain this containment and a presentation of the final solution is given in Section D of Chapter III. We now discuss the test cases used to determine which of those combinations gives the most accurate simulation.

The first test case that revealed a problem with the implementation of the boundary condition was a Gaussian, of the form in (65), with a significant nonzero initial momentum inside the infinite square well. Figure 13 shows the progression of a successful bounce.

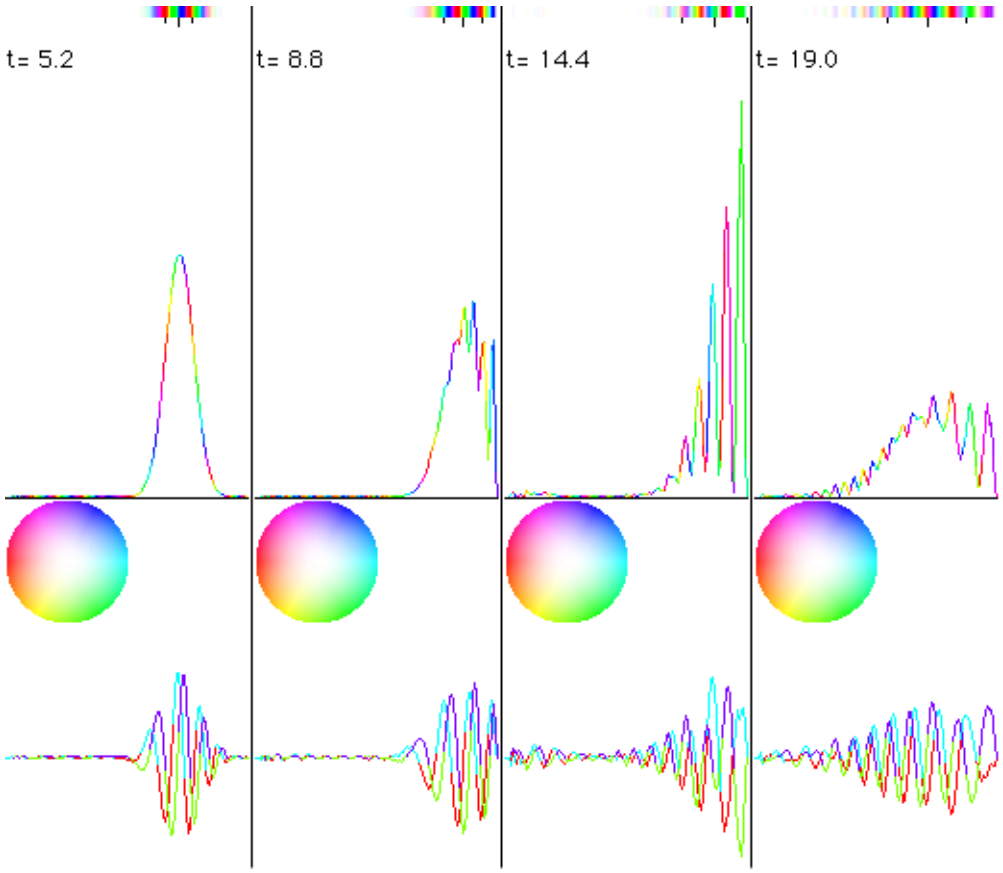


Figure 13. A moving Gaussian bouncing off a wall of the infinite square well.

Theoretically, the wavefunction should encounter the wall of the well, then bounce back without losing energy. The initial simulations showed a preservation of total energy until the time step that the probability density of the wavefunction at the wall became significant, after which the wavefunction suffered a measurable energy loss. Since the energy loss only began when the wavefunction "touched" the wall, this evidence strongly suggested that incorrectly defined boundary conditions caused the loss.

More tests were run using other combinations of boundary condition

parameters while bouncing the Gaussian off the wall. The prescription of this test relied on observing a decrease in the energy diagnostic after the bounce. While this method did rule out combinations that resulted in gross losses, it could not distinguish among the combinations that were "close" to being correct. It was realized that the energy diagnostic itself is also not clearly defined at the boundary for some of the same reasons this investigation was underway. So it became clear that this test was not sufficient, and a new method of testing had to be found.

Utilizing the eigenstates of the infinite square well became the next choice to rigorously test the boundary condition methods. The behavior of these eigenstates relies on the properties of both boundaries simultaneously, so any "bleeding" of energy due to the boundaries should be clearly evident. If we can gain confidence in an accurate simulation of these eigenstates, then it seems plausible, given the behavior seen in the SHO case, that any superposition of these eigenstates will also be correct.

The properties of these eigenstates are fairly straightforward. The eigenstates are described by

$$\psi_n(x) = \sqrt{\frac{2}{L}} \sin\left(\frac{n\pi x}{L}\right) \tag{73}$$

where n is the quantum number of the eigenstate, a positive integer, and L is the width of the well. As in the SHO, an eigenstate's phase will evolve at a

particular frequency proportional to the energy eigenvalue of the state, which takes the value

$$E_n = \frac{h^2 n^2}{8mL^2}, \tag{74}$$

and, obviously, remains constant throughout the existence of the state.

Early tests easily showed how incorrect boundary conditions presented problems maintaining the eigenstate. Most of the incorrect possibilities introduced kinks at both edges of the wavefunction that propagated inwards. The kinks would increase in strength and in number, eventually dominating over the wavefunction entirely. Eigenstates with n=1 through n=11 were used, and this behavior was seen in all such cases. These tests eliminated the wavefunction extrapolation and virtual classical particle phase adjustment candidates discussed in Section D of Chapter III. It was found that modest guard cells in which the wavefunction was zeroed were needed, and the virtual classical particles were indeed reflected (without which the wavefunction simply disappears). Some of these eigenstates are shown in Figure 14.

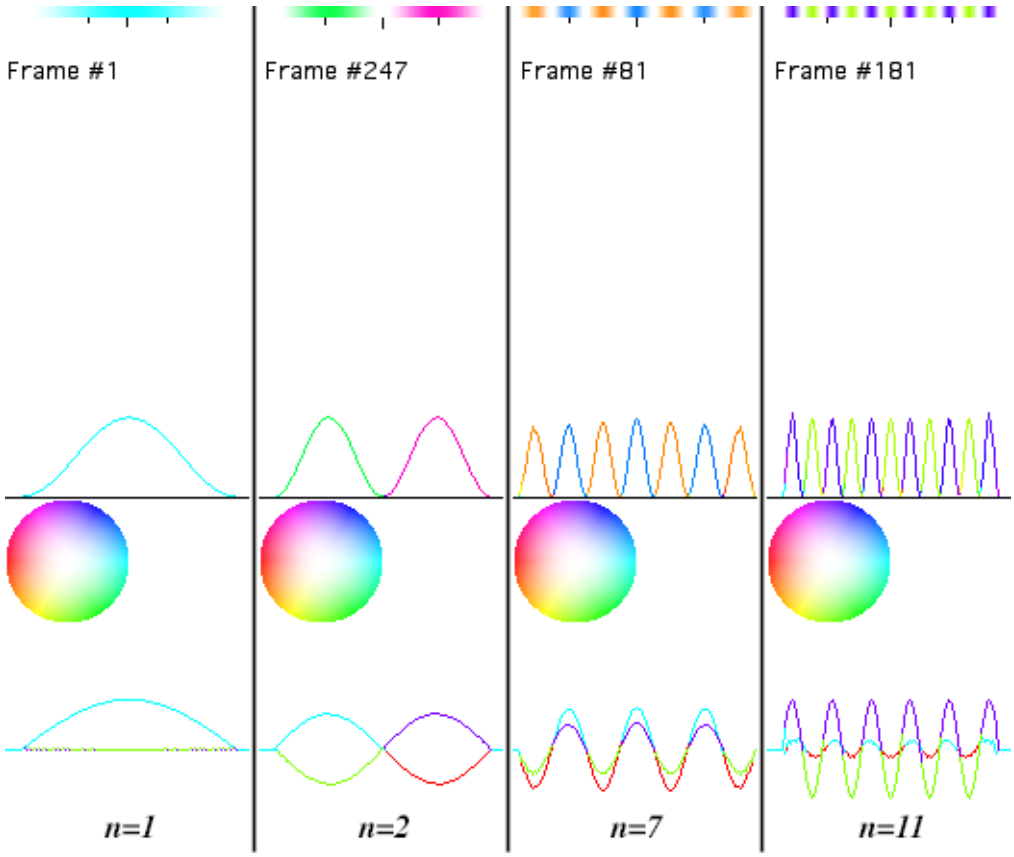


Figure 14. Example eigenstates of the infinite square well. Frames from the n=1, n=2, n=7, and n=11 cases are shown.

Upon establishing the grids where the wavefunction is to be zeroed, that defined the walls of the well and, consequently, the width L of the well. The next question is: where should the virtual classical particle be reflected? Our first hypothesis was at the walls, in particular, the grid points where the wavefunction becomes zero. This hypothesis was attempted, and qualitative properties, such as their long-term stability and evidence of phase evolution, of the eigenstates were preserved.

However, upon close inspection of the precise rate of its phase evolution,

it became clear that the eigenstate was not behaving as if the well was of width L but, instead, of width $L-1$. The precise positions of boundaries and particle reflections were carefully rechecked, and the result was confirmed.

Other experiments were attempted. When the left ($x=0$ boundary) particle reflection was displaced left by one, the eigenstate behaved as if the well expanded by one. Likewise, when the right ($x=L$ boundary) particle reflection was pushed right by one, the eigenstate behaved as if the well expanded by one. Finally, when the reflection points were each placed one-half grid point beyond their respective wall, the eigenstates behaved as if they were in a well of width L, making this measurement consistent with the positions of the explicitly zeroed wavefunction.

This behavior was independent of all other computational aspects of the code. It was consistent in all observed eigenstates and was seen when the number of grid points in the simulation was adjusted arbitrarily. It seemed that, when the reflection points are close to the edge of the wavefunction zeroing, the precise behavior of the wavefunction are more dependent on the reflection points rather than the zeroed edge of the wavefunction. In addition, this dependence shows the state responding to a wall one-half grid point closer than the reflection point. While this empirical result was a surprise to us, this combination of parameters also allowed the Gaussian to bounce off the wall without a measurable loss of energy.

Besides determining the boundary conditions necessary to maintain a state in an infinite square well, this investigation shows how precisely properties of the quantum system can be measured. The numerical values of L in this investigation were 112, 120, 124, 248, and 252. By observing the frequency of the eigenstate oscillation, it was possible to distinguish between a well of width 252 versus a well of width 251, or 251.5, while varying a range of other independent parameters. Such precise determinations should provide support for this code's utility and robustness.

E. Barriers

Other attempts at duplication of well-known quantum problems were made. The finite square well, and a variety of quantum barrier problems were attempted. For example, the square barrier potential,

$$V_{squarebarrier}(x) \equiv \begin{cases} V_0, \textit{ if } 0 < x < w \\ 0, \textit{ elsewhere} \end{cases}, \tag{75}$$

was implemented as shown in Listing 7.

```
parameter(width = 4, height = 16.0)

                        . . .

do k=1,nblok
  joff = noff(k) - 2
  do j=1,nxpmx
```

```
!            ! Rectangular barrier or well
            xt = (j + joff - nx/2)
            if (abs(xt).le.width) then
             if (abs(xt).gt.(width-2)) then
              if (xt.lt.0) then
                pt(j,l,k) = pt(j,l,k) + &
     &                adjustment*height*0.5*(xt+width)

                fx(j,l,k) = fx(j,l,k) -
adjustment*height*0.5

              else
                pt(j,l,k) = pt(j,l,k) + &
     &                adjustment*height*(width-xt)

                fx(j,l,k) = fx(j,l,k) +
adjustment*height*0.5

              end if
             else
                pt(j,l,k) = pt(j,l,k) + adjustment*height
             end if
            end if

          end do
        end do
```

Listing 7. Code sample implementing a (nearly) rectangular well. Because a value of the derivative of the potential is required to make the force array consistent with the potential array, the code cannot handle a potential with sharp discontinuities. Consequently, this code implements a barrier potential with very steep edges.

nx, the width of the simulation, is used to center the barrier.

The problem with implementing this barrier is that this potential possesses sharp boundaries. Such properties contradict a basic assumption of the code that the grid points are sufficiently fine to resolve all features of interest as continuous functions. Consequently it was little surprise that quantitative measurements (specifically, transmission and reflection

coefficients) on simulations resulting from using this potential did not precisely match theory, but much of the qualitative features were clearly evident. A demonstration of quantum tunneling, with partial transmission and reflection, is shown in Figure 15.

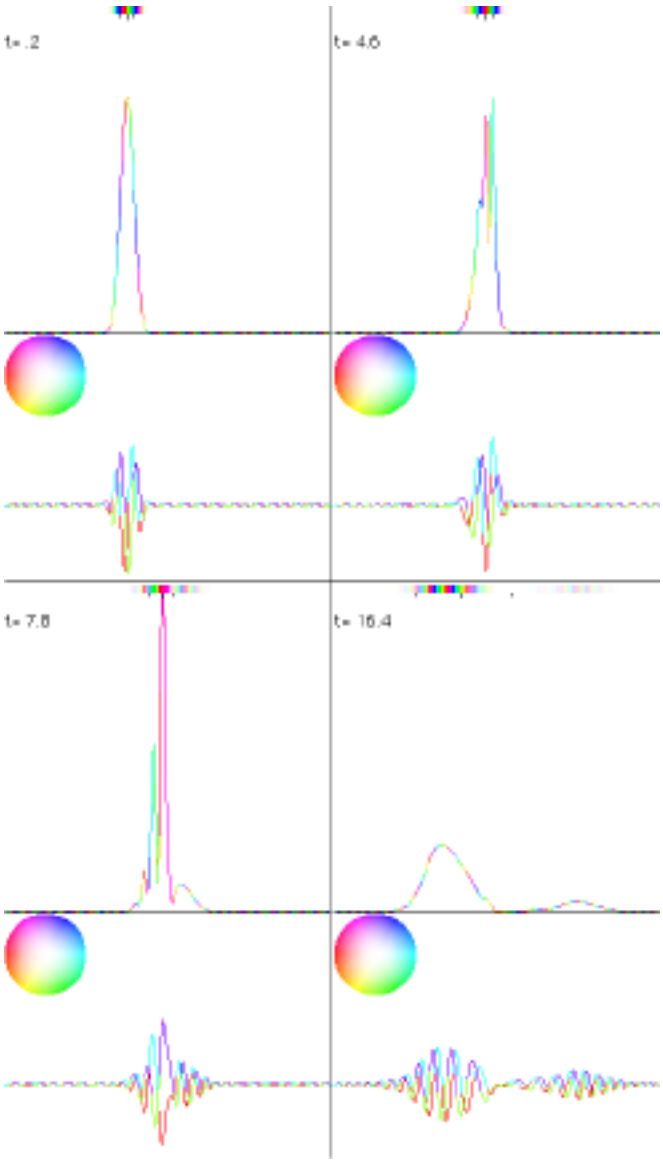


Figure 15. Evolution of a Gaussian wavefunction colliding with a square barrier eight grids wide in the center (not drawn). The energy of the Gaussian is just enough for a significant amount of transmission and reflection, seen in the last

frame.

It may be possible to implement the ideal rectangular potential by customizing elements of the particle pusher in a fashion similar to the boundary condition implementation used for infinite square well. These possibilities will be discussed in Chapter VII - Future Work.

F. Fermion Statistics

Throughout the process of building the code, the routines, arrays, and loops were designed to handle multiple interacting particles. Mixed with the above tests were ones involving up to sixteen particles, but the earliest experiments of significance on more than one quantum particle involved quantum particle statistics. The particular category of quantum statistics that we wanted to address first was the statistics of fermions since we ultimately wish to apply this code to modeling electrons, categorized as fermions. The "holy grail" would be to model the full multiparticle wavefunction, however, the memory requirements to store, in the position representation, such a wavefunction scales exponentially as a function of the number of quantum particles. The memory available in today's largest computers would limit the model to a half-dozen quantum particles.

Therefore, it is in our interests to determine if there exist alternative

means of modeling fermionic behavior and the extent of their validity. We use the following approach. During the simulation, we model the quantum wavefunctions assuming they are representable as in (54). Interactions use a "mean-field approximation". As a post-processing step, we build the antisymmetrized multiparticle wavefunction using the data set generated by the simulation. The first case we will present involves two electrons, so their antisymmetrized wavefunction would be

$$\psi_{12}(x_1, x_2, t) = \left(\psi_1(x_1, t)\psi_2(x_2, t) - \psi_2(x_1, t)\psi_1(x_2, t)\right) N_{12}(t) \tag{76}$$

where $N_{12}(t)$ is a normalization factor such that $\langle \psi_{12}(t) | \psi_{12}(t) \rangle = 1$, for all t.

Diagnostics and tests of interest would then be performed on ψ_{12}. For example, the correlation calculation would be

$$c(\tau) = \int \langle \psi_{12}(t+\tau) | \psi_{12}(t) \rangle \, dt \tag{77}$$

or, in the position representation,

$$c(\tau) = \iiint \psi_{12}^*(x_1, x_2, t+\tau)\psi_{12}(x_1, x_2, t) \, dx_1 dx_2 dt \tag{78}$$

As before, the Fourier transform of c should give the energy spectrum of ψ_{12}.

The first system studied using this approach was two fermions in an infinite square well. Their electrostatic interactions were turned off so we could focus on behavior involving the statistics of the system. A fundamental aspect of their behavior we wanted to observe was the Pauli Exclusion Principle,

where no two fermions in a given system had the same quantum number.

Two sets of initial conditions were used. The first loaded the wavefunction arrays each with an arbitrary superposition of the five lowest eigenstates of the infinite square well. The phases of the coefficients of the eigenstates were different for each particle. The second initial conditions represented a pair of arbitrarily-chosen, low-energy Gaussians, beginning in different parts of the well with opposite initial momenta. While designed to resonate the lowest energy eigenstates, these Gaussians were an arbitrary choice indeed. Frames from these simulations are shown in Figure 16.

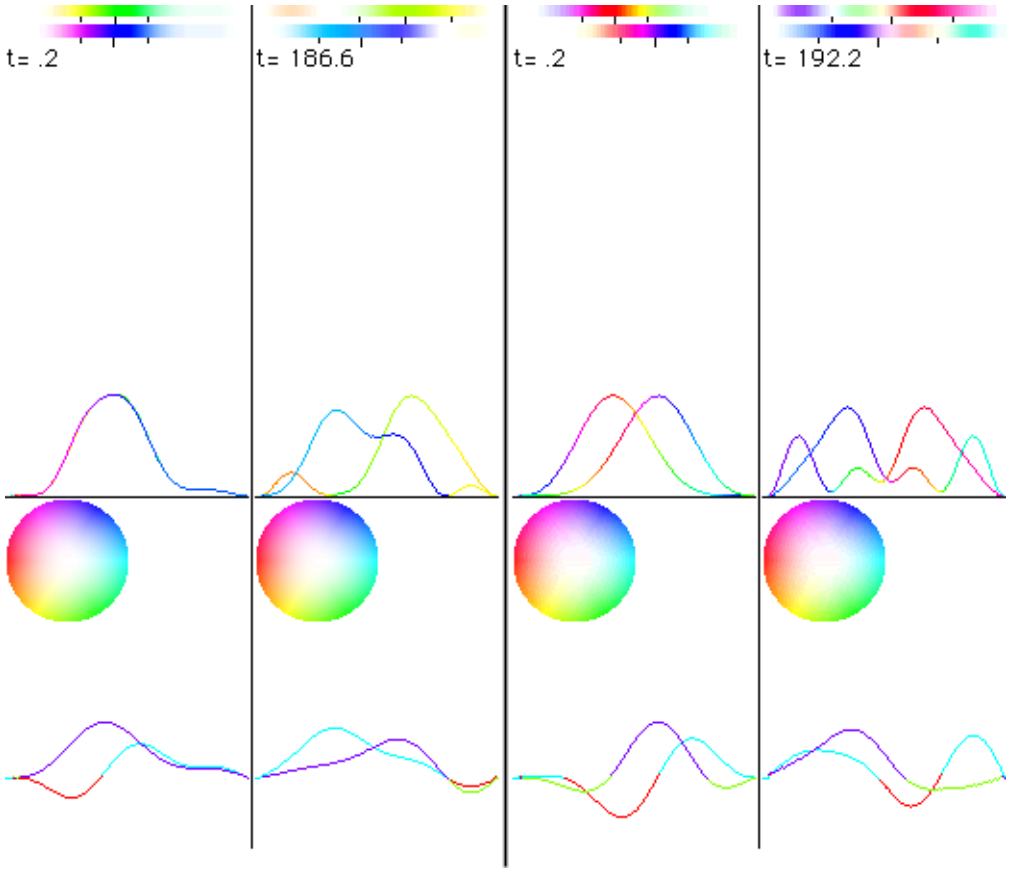


Figure 16. A pair of frames each from a pair of wavefunctions in an infinite

square well. The left frames are a run using a superposition of eigenstates, while the right frames are from a run using arbitrarily chosen Gaussians.

A new code was constructed to calculate ψ_{12} and its energy spectrum from the $\psi_1(x_1,t)$ and $\psi_2(x_2,t)$ data generated by the quantum PIC code. It was found that the most efficient manner to calculate the Fourier transform of c was to reinterpret (78) as a convolution in time. Applying the convolution theorem to (78) yields

$$\tilde{c}(\upsilon) = \iint \tilde{\psi}_{12}{}^*(x_1, x_2, \upsilon)\tilde{\psi}_{12}(x_1, x_2, \upsilon)\, dx_1 dx_2 \qquad (79)$$

where $\tilde{c}$ and $\tilde{\psi}_{12}$ are the Fourier transforms in time of c and ψ_{12}, respectively. With regards to computation, this form suggests that a Fourier transform in time, rather than many integrals in time (suggested by (78)), should be performed on ψ_{12} first, then the integrals over space are performed. Using the FFT reduces the computation time from O(N²) to O(N lg N), where N is the number of time steps.

The correlation code was carefully designed to efficiently handle this non-trivial problem. The runs described here are 128 grid points wide and over 65536 time steps in length. While the output of the quantum PIC code was 128 MB in size, the resulting data set representing the complete time sequence of ψ_{12} in double-precision was 16 GB in size. 16384 FFT calls on this data set were needed. This correlation code was designed to efficiently utilize processors in

parallel. Initialization involved distributing the 128 MB data file across the processors. Given a particular value of x_2, the processors generated a double-precision form of one slice, identified by x_2, of ψ_{12} at a time. ψ_{12} was calculated for all t in each processor while partitioned along x_1 between processors. The FFTs in t were performed, then the integral along x_1 in each processor was calculated and accumulated for each frequency, after which the code directed the processors to move on to the next value of x_2. Once the accumulation finished for all x_2, the correlation data as a function of frequency was accumulated between processors, the final data set was saved to disk, and the code ended.

The energy spectrum of one particle eigenstate in the infinite square well is given by (74). Therefore, the frequency of oscillation for an eigenstate in this two particle system is given by

$$\upsilon_{n,m} = \frac{h^2}{8mL^2}\left(n^2 + m^2\right) \tag{80}$$

where n and m are the two integer quantum numbers of the two-particle state. We would expect to find peaks consistent with (80), however, if this system obeys the Pauli Exclusion Principle, we should see none where n and m are equal.

The energy spectrum resulting from the five eigenstate case is shown in Figure 17.

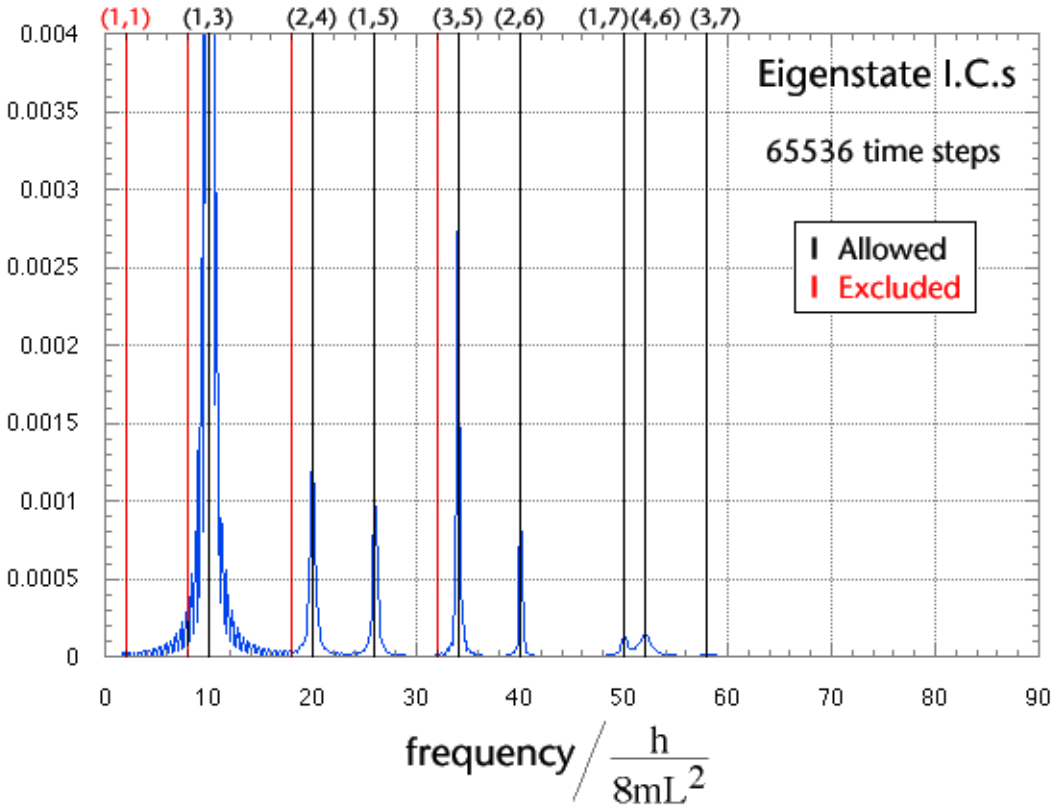


Figure 17. Energy spectrum from the evolution of a fermion pair initialized using the five lowest energy eigenstates of the infinite square well. The frequencies marked in red are those disallowed by the Pauli Exclusion Principle.

The blue graph is the energy spectrum. The base of the lowest peak shows spurious noise because it is very strong and has finite width.

We find four points of good news. First, we do see peaks, which means that a discrete finite energy spectrum is evident in the system, which is plausible given that these effects are supposed to be quantum. Second, we see that the peaks are well aligned and consistent with the frequencies predicted by (80), which are marked with black lines. Third, the peaks we do see are at frequencies we would expect, that is, where the quantum numbers are different

(e.g., (1,3), (2,4), (1,5), and (3,5)) and in the range we would expect (between states 1 and 5, inclusive). Fourth, we do not see peaks where we do not expect them, in particular, where the quantum numbers are the same ((1,1), (2,2), (3,3), and (4,4), marked in red. Note that (5,5) is degenerate with (1,7).). This finding is important because this absence is consistent with the Pauli Exclusion Principle, which was a phenomena that we intended to duplicate.

We come to the, not bad, but unexpected news. There are some peaks that we were hoping to see (such as (1,2)), but did not. This could be explained because the choice of the coefficients was arbitrary, so it was not clear that such states would appear. In addition, the spectrum shows particular peaks ((2,6), (1,7), (4,6), (3,7)) that are unexpected. While allowed by the Pauli Exclusion Principle, these frequencies are not directly explained by the eigenstates input into the code. Their presence may be explained by small differences between the "ideal" set of the eigenstates and the actual numerical, discretized wavefunction loaded as initial conditions into the wavefunction arrays. They could also be noise or numerical error developed as the wavefunction evolved. In either case, their magnitude is quite small compared to the primary peaks at the expected frequencies.

Figure 18 shows the energy spectrum resulting from using a pair of arbitrarily selected Gaussians as initial conditions.

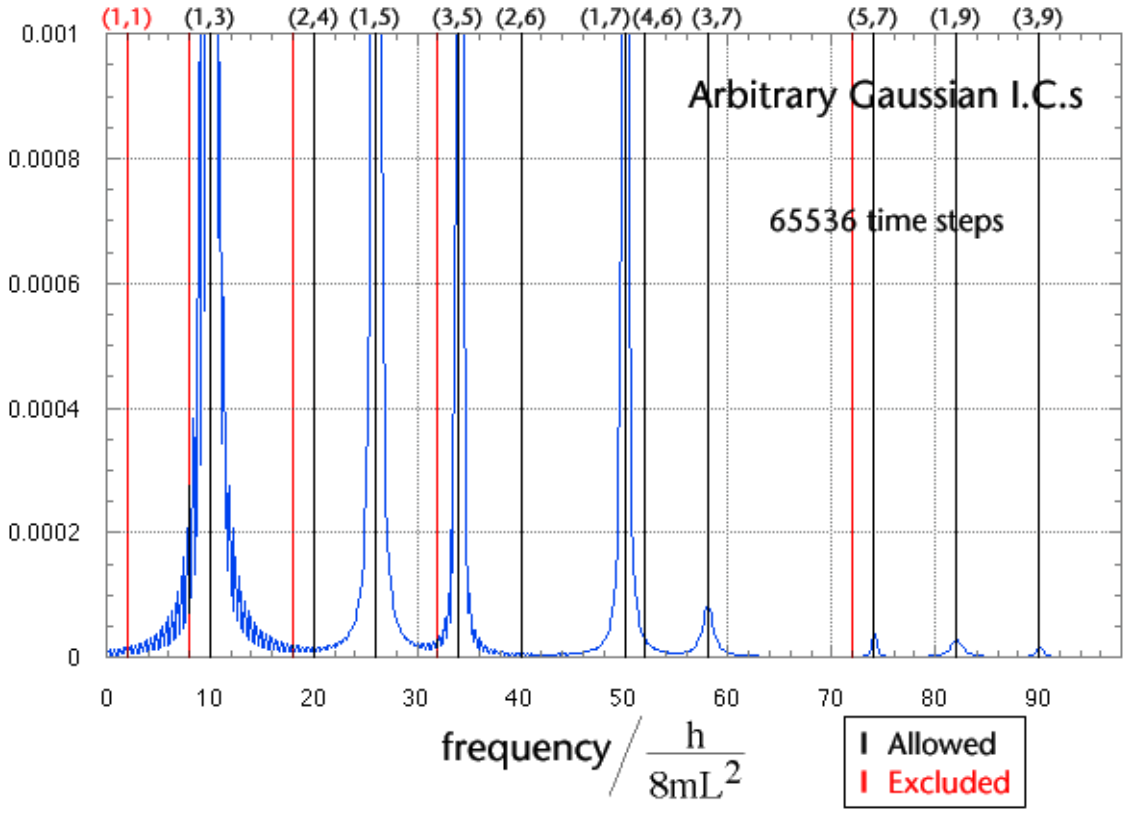


Figure 18. Energy spectrum from the evolution of a fermion pair initialized using arbitrarily chosen Gaussians.

The good news is, again, we find plenty of peaks where they should be, and find no peaks where the should not be (where $n = m$). We also observe much higher energy states ((5,7), (3,9)), than we saw in the previous spectrum. Although not identical, we observe much of the same energy structure using these initial conditions as we did using much more carefully chosen initial conditions.

The results seen in this chapter provide support for the utility of these codes. The observation using the arbitrarily chosen Gaussians is significant

because it tells us that it is not necessary to know the spatial or energetic structure of a system *a priori* in order to use these modeling and analysis techniques to extract useful information (such as eigenfrequencies) about the system. Inputting low-energy Gaussians is sufficient to excite the lower, although not necessarily the lowest, energy eigenstates of the system. Experimentation given such a system becomes a simple procedure of choosing modestly judicious initial conditions and running the codes. The level of flexibility of the code makes the above possible. And, as we can see in the energy spectra, the noise in the data is very small, while the strength and clarity of the peaks are very significant. These characteristics indicate the high quality of the simulation and analysis. The high flexibility and high quality of the code's simulations have significant utility when we wish to better understand quantum systems we encounter, especially those about which we know little.

V. The One-Dimensional Atom

A. The Problem

Our next system of interest is the one-dimensional atom. We define the one-dimensional atom as a number of fermionic electrons bound to a nucleus with a charge equal in strength yet opposite to that of the sum of the electrons' charge. The spatial dependence of the nuclear charge is defined by

$$V_{nucleus}(x) \equiv \lambda|x| \tag{81}$$

This problem was chosen for because it is analytically difficult while being compatible with the quantum PIC code, and it features many of the technical challenges that this code is intended to address. In addition, it allows us to investigate behavior known to occur in its three-dimensional counterpart.

B. The One-Electron Case

We attempted a theoretical analysis of one electron in a one-dimensional atom to estimate the energy of the ground state. We first used a Bohr-Sommerfield quantization ("p dq") method to estimate the ground state energy, but the lowest result found was derived using a variational approach with a Gaussian wavefunction of the form described by (65)

$$\psi(x,t=0) = \frac{1}{\sqrt{\sigma\sqrt{2\pi}}} \exp\left(-\frac{(x-x_0)^2}{4\sigma^2}\right) \exp\left(i\frac{p_0 x}{\hbar}\right) \tag{65}$$

with p_0 set to zero and x_0 centered on the nucleus. Evaluating the energy of this wavefunction with the potential described by (81), then finding the minimum of the energy as a function of σ, gave the following estimate for the ground state energy

$$E_0 \approx \frac{3}{2}\sqrt[3]{\frac{\lambda^2\hbar^2}{2\pi m}} \tag{82}$$

Implementing the one-dimensional atom potential in the code was straightforward. The external potential described by (81) was implemented using a loop in the external potential routine shown in Listing 8.

```
real :: slope
parameter(slope = 1.0/4.0)

                    . . .
```

```
      do k=1,nblok
        joff = noff(k) - 2
        do j=1,nxpmx

!           ! 1-D atom potentials
            xt = j + joff - nx/2
            if (xt.lt.0) then
              pt(j,l,k) = pt(j,l,k) - adjustment*slope*(xt)
              fx(j,l,k) = fx(j,l,k) + adjustment*slope
            else if (xt.gt.0) then
              pt(j,l,k) = pt(j,l,k) + adjustment*slope*(xt)
              fx(j,l,k) = fx(j,l,k) - adjustment*slope
            end if

        enddo
      enddo
```

Listing 8. Code listing that implements the one-dimensional atom potential. `xt` is the position relative to the center of the potential.

Setting the potential's `slope` to 1/4 was found to provide stable simulations while providing adequate confinement for systems that we found interesting and fit within a 128 grid point space. A variety of initial conditions based on (65) were attempted for these early experiments. Naturally, the electron charge was set to match the magnitude of the charge implied by `slope`.

The initial conditions that were attempted involved a Gaussian of the form of (65) with zero initial momentum, a standard deviation given by the calculation leading to (81), and x_0 offset from the center of the potential. The offset was provided to excite the higher states in addition to the ground state. Frames from this simulation are shown in Figure 19.

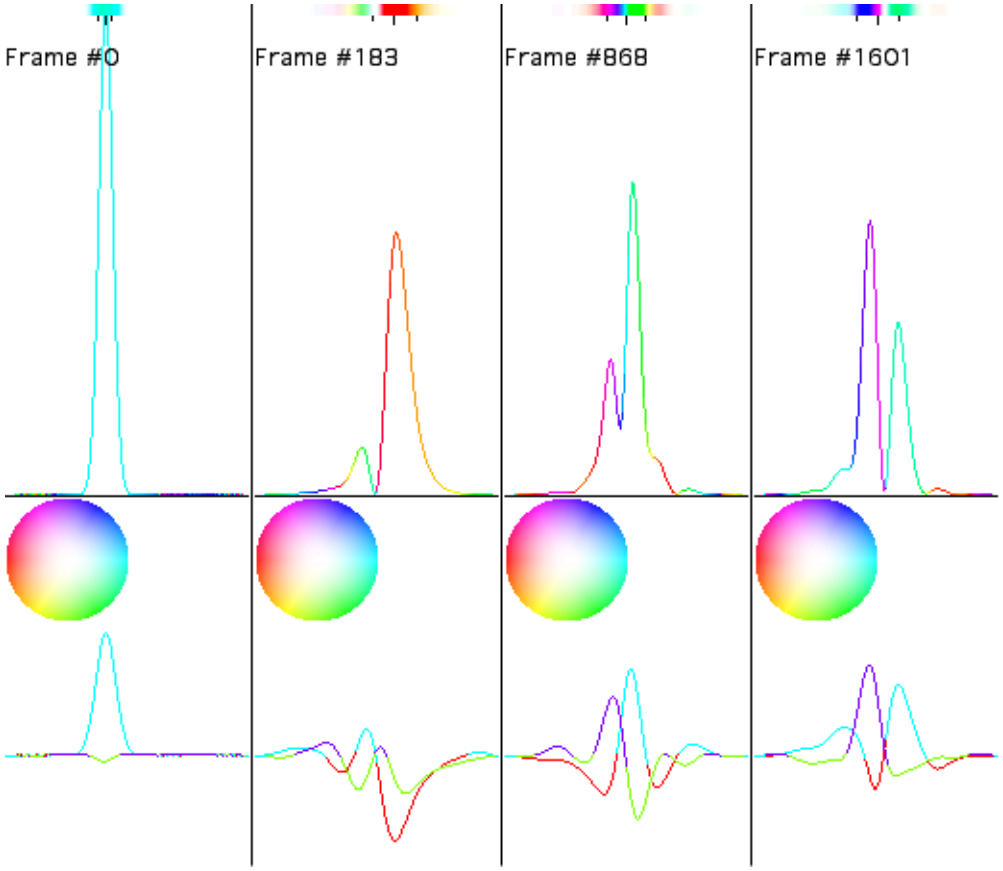


Figure 19. Frames from the evolution of an electron bound to a one-dimensional atom.

The result of performing a correlation calculation, as described in Section C of Chapter IV, on this simulation is shown in Figure 20.

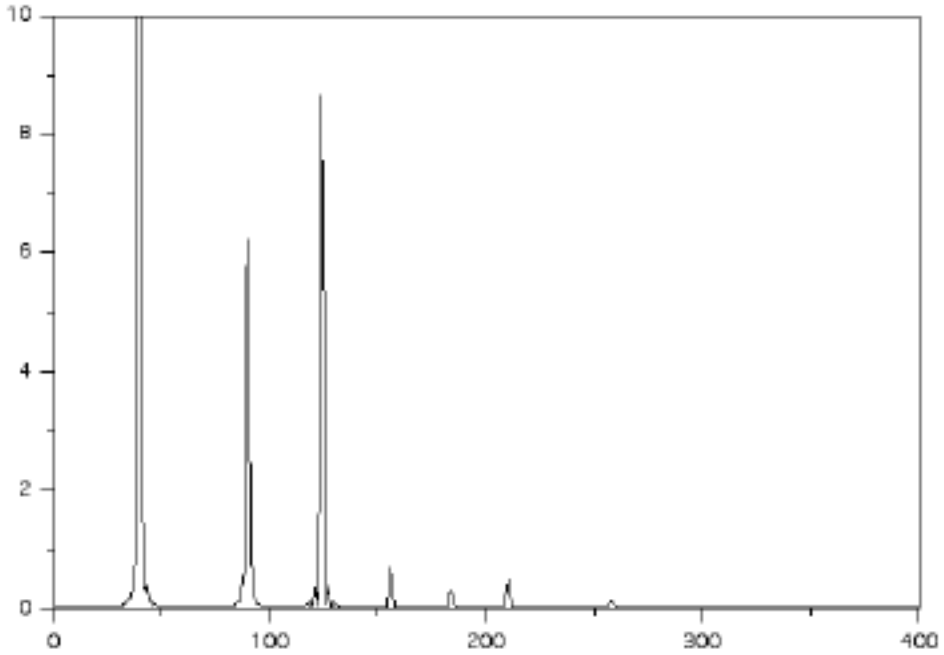


Figure 20. Energy spectrum of a simulation shown in Figure 19. The horizontal axis is mode number, which is proportional to the frequency of oscillation of an eigenstate of the system. This spectrum was derived from 8192 time steps of the simulation.

The ground state frequency seen in the simulation was very close to that predicted by (82). The frequency corresponding to (82) translates to mode number 38.8 in the graph of Figure 20. Interpolated between points, the lowest mode is observed to be at 38.55, just under the prediction using (82). Excited modes seen are estimated to be at 88.55, 123.15, 154.85, 182.55, and 209.35. Analyzing simulations using alternative initial conditions also show excited states at these mode frequencies.

The discrepancy between (82) and the observed ground state energy is very plausible. (82) was an estimate based on a variational method, and would

provide an upper bounds to the ground state energy, while the actual ground state energy can be even lower. The simulation's prediction, being close to the analytical prediction but lower, is consistent with the analysis. The closeness of these results lends support to the correctness of the code. The code is also able to easily show other energy eigenvalues, showing quantitative measurements of the one-dimensional atom's energy structure using very little analytical work. The computational requirements for this simulation are very modest: a few hours on a single modern personal computer.

C. The Two-Electron Case

The next case of interest was the two-electron one-dimensional atom. We are guided by the methods explored involving the two fermion infinite square well, described in Section F of Chapter IV. We assume that we may choose the initial conditions to be a subjective estimation of the states we expect to find, as indicated by the aforementioned investigation. We present our choice of a pair of Gaussians for the pair of electrons: one is placed in the center of the well, and the other is offset from the center by approximately four standard deviations of the Gaussians. The intention is to excite the lowest modes of the two-electron system. The charge on the electrons is set to half that of the one-electron case, and they are allowed to interact with each other

electrostatically. Frames from their evolution is shown in Figure 21.

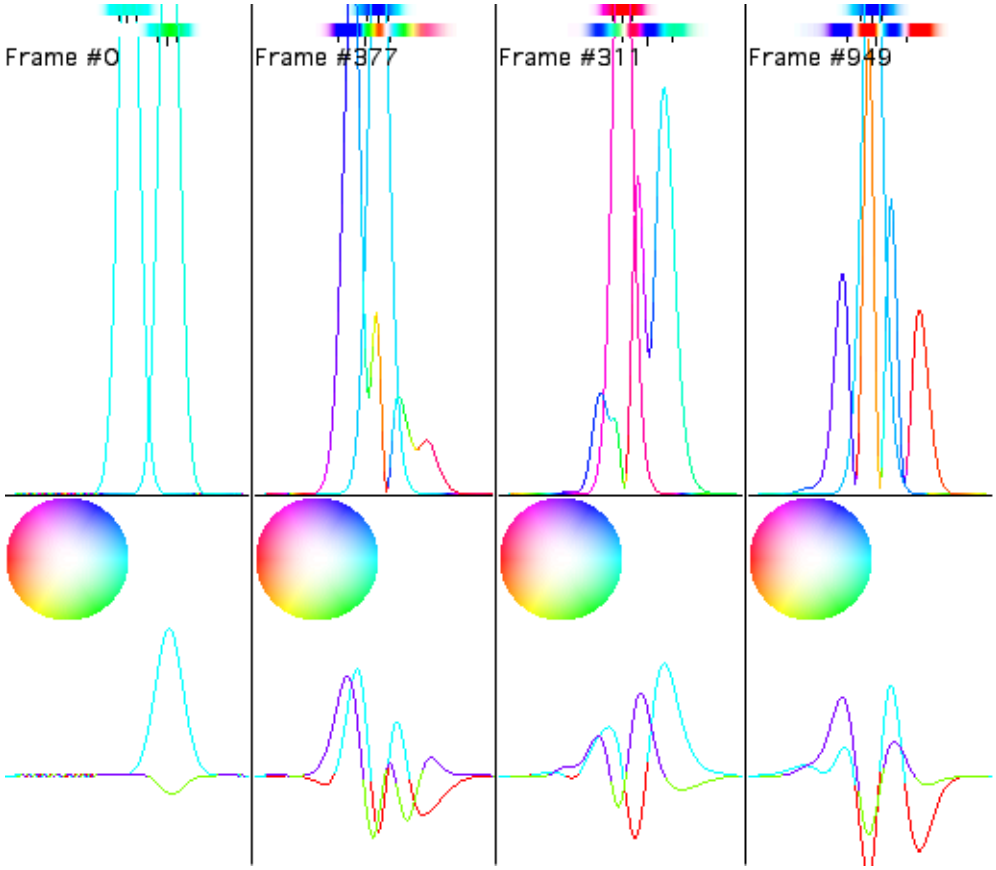


Figure 21. Four frames from two-electrons bound to a one-dimensional atom.

In this figure, the two wavefunctions are shown in two color bars at the top and superimposed in the middle graph. The lowest graphs show the real and imaginary parts of the second wavefunction. The first wavefunction is begun centered in the well, and the second is offset. Note how the second wavefunction oscillates around the first, which is nudged by the second.

After the simulation was complete (taking less than a day on a single

processor computer), the data was analyzed as prescribed in Section F of Chapter IV, in particular computing (76) and (79) as described in that section. The resulting energy spectrum is shown in Figure 22.

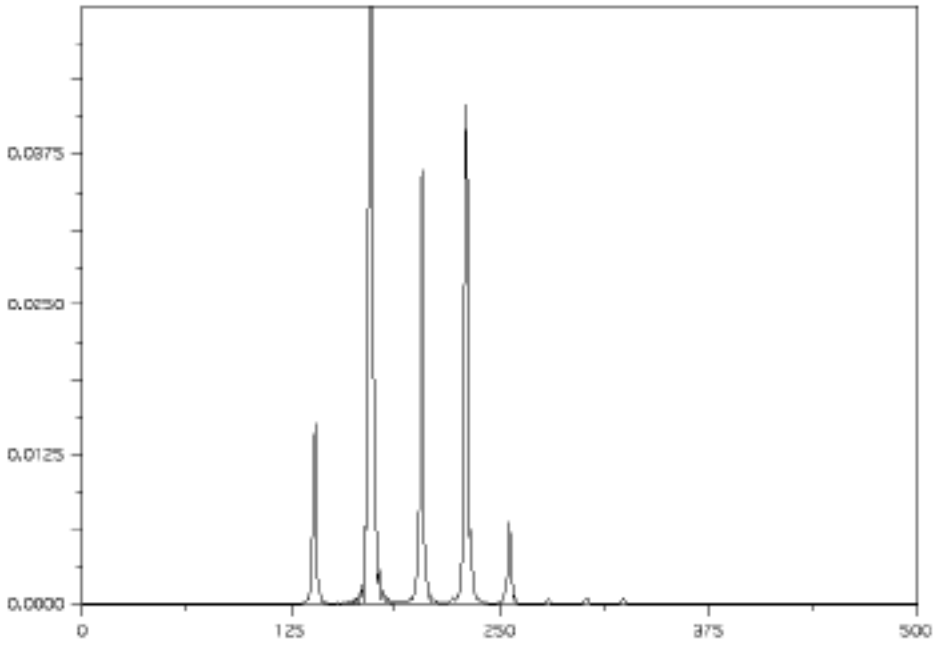


Figure 22. Energy spectrum of the fermionic two-electron simulation shown in Figure 21. The horizontal axis is mode number, which is proportional to the frequency of oscillation of an eigenstate of the system. This spectrum was derived from 8192 time steps of the simulation.

The resulting energy spectrum shows five well-resonated modes and three additional modes. Note that the energy of the lowest peak has a mode number that is slightly greater than the sum of the lowest two modes seen in the one-electron case (138.55 > 38.55 + 88.55). The discrepancy is most likely due to the mutual electrostatic repulsion of the electrons. This seems plausible as a "(1,2)" state, and indicates consistency with the Pauli Exclusion Principle. With one run,

we are able to clearly observe a portion of the energy structure of a two-electron fermionic system. Again, at a several hours per simulation run, and less than an hour for the analysis, the computational demands are modest.

D. Eigenstate Extraction

In the course of writing the correlation code that computes (79), it was discovered that additional important information about the system can be generated from the same analysis. A portion of this discussion is similar to one independently conceived by Neuhauser [58] and Decyk [59], but the regime of the application is different. Consider the Fourier transform in time of the wavefunction

$$|\tilde{\psi}(\upsilon)\rangle \equiv \int |\psi(t)\rangle \exp(2\pi i \upsilon t)\, dt \tag{83}$$

Applying the convolution theorem to (71) gives

$$\tilde{c}(\upsilon) \equiv \langle \tilde{\psi}(\upsilon) | \tilde{\psi}(\upsilon) \rangle \tag{84}$$

$\tilde{c}(\upsilon)$ is simply the inner product of $|\tilde{\psi}(\upsilon)\rangle$ with itself, essentially describing "how much" of the wavefunction is oscillating at a particular frequency υ.

For the sake of simplicity, consider what happens when $|\psi(t)\rangle$ is an energy eigenstate. Suppose $|\psi(0)\rangle$ is an energy eigenstate $|e_n\rangle$ with energy

eigenvalue E_n. Therefore, according to (1),

$$|\psi(t)\rangle = \exp(-\frac{iE_n t}{\hbar})|e_n\rangle \tag{85}$$

Consequently, with (83),

$$|\tilde{\psi}(\upsilon)\rangle = \delta(\upsilon - \frac{E_n}{h})|e_n\rangle \tag{86}$$

which results, using (84), in a peak in $\tilde{c}$ at $\upsilon = \frac{E_n}{h}$. But (86) also says that $|\tilde{\psi}(\upsilon)\rangle$ contains a description of the energy eigenstate $|e_n\rangle$ at that same frequency.

As an interim step to computing $\tilde{c}(\upsilon)$, the correlation code has already been computing $|\tilde{\psi}(\upsilon)\rangle$ in the position basis. We can serendipitously use the existing structure of this code in the following way. If we save a portion of $|\tilde{\psi}(\upsilon)\rangle$ at the right point in the code, we can extract an *entire energy eigenfunction* for every energy eigenvalue we identify in $\tilde{c}$.

Another conceptual approach to this technique is an analogy to tuning a radio. Searching through the frequency spectrum is like tuning the radio. Locating and selecting a peak is like tuning the radio to a particular station, recognized by its signal strength. The signal is modulated at that frequency, and the radio is designed to demodulate the signal at its carrier frequency and provide that station programming. Likewise, the kernel in (83), given the right frequency, will compensate for the inherent oscillation frequency of the

eigenstate, and provide the desired signal.

The correlation code, as described in Section F of Chapter IV, computes $|\tilde{\psi}(\upsilon)\rangle$ in slices, so the task became saving the desired slices of the Fourier transformed data set (32 MB for the one-electron case; 16 GB for the two-electron ψ_{12}) that represents $|\tilde{\psi}(\upsilon)\rangle$, and then collating the pieces into a format convenient to display. The result of the one-electron case at five of the stronger peaks seen in Figure 20 are shown in Figure 23.

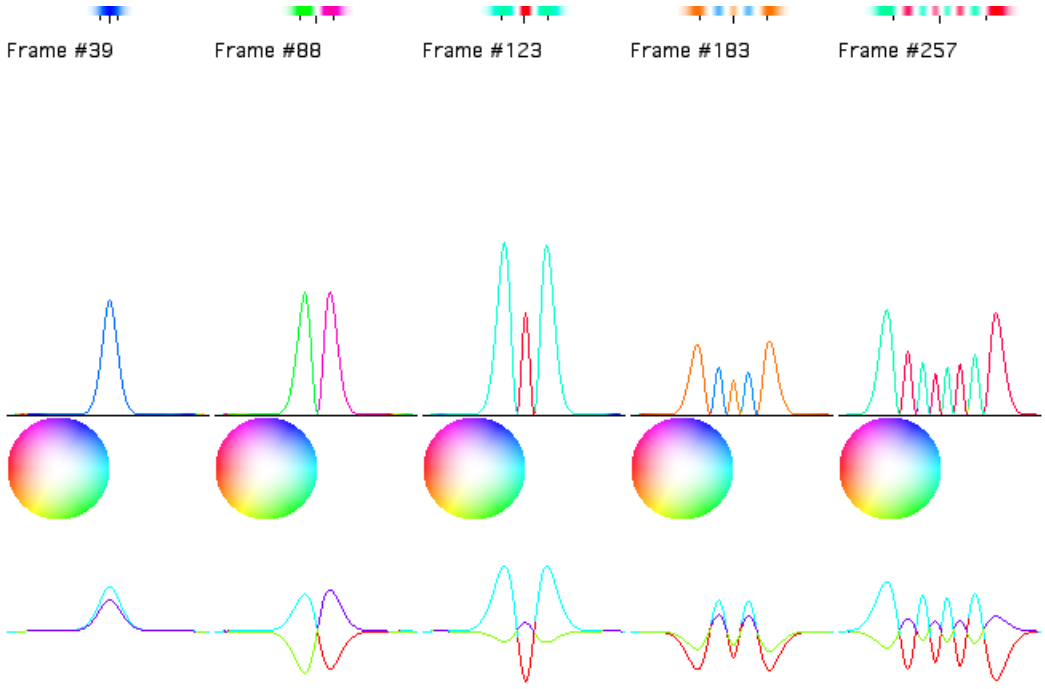


Figure 23. Five energy eigenstates extracted from the one-electron one-dimensional atom simulation seen in Figure 19. Note similarities of these states to the n=0, 1, 2, 4, and 7 eigenstates of the simple harmonic oscillator.

Given our experience with the SHO eigenstates, the eigenstates extracted from the one-electron simulation show spatial structure that we would find

plausible for the one-dimensional atom. Since the extraction shows that these states are those that oscillate with a particular frequency in the time evolution of a one-electron $|\psi\rangle$, we propose that these are the energy eigenstates of the one-dimensional atom. Their similarity to the SHO eigenstates also gives confidence in proposing specific quantum numbers to each eigenstate and, correspondingly, the peaks seen in Figure 20. Simulation runs using variations on the initial conditions used in this example result in a different distribution of peaks, allowing us to easily locate and extract other eigenstates that interest us.

The result of the eigenstate extraction on the five lowest peaks, seen in Figure 22, of the fermionic two-electron one-dimensional atom is shown in Figure 24. The implementation of this extraction was much more involved than in the one-electron case and was first tested using the data sets generated for the infinite square well fermion runs described in Section F of Chapter IV.

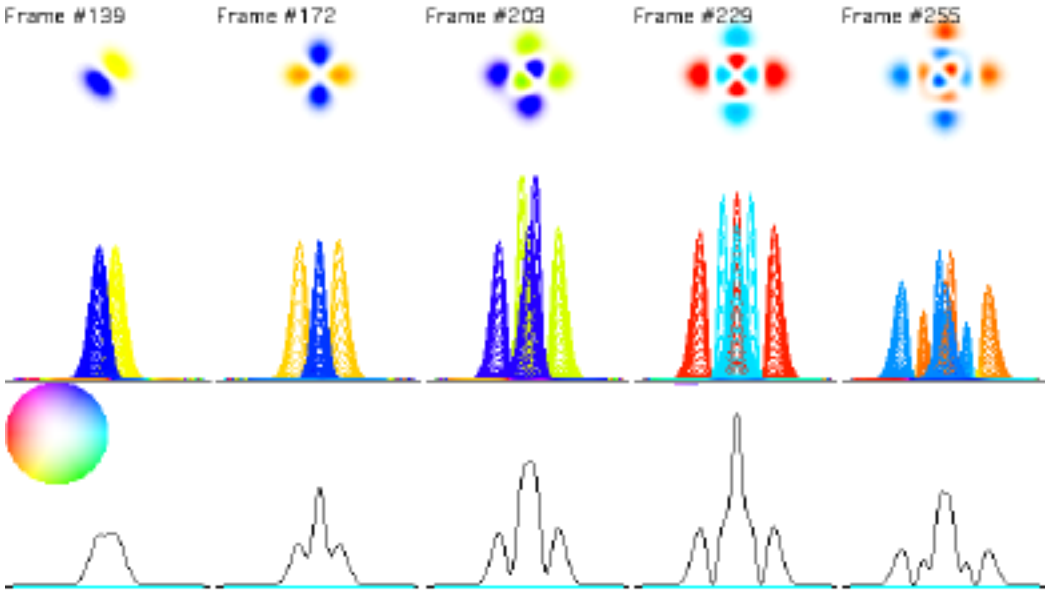


Figure 24. Five two-electron fermionic eigenstates of the one-dimensional atom.

This visualization possesses some differences from previous figures, so its components will be described here. The phase-color convention is as it was described for Figure 8. The top pictures plot $\psi_{12}(x_1, x_2)$'s phase as color and its magnitude squared as the strength of that color, with the x_1 and x_2 plotted horizontally and vertically. The middle graph plots $\psi_{12}(x_1, x_2)$ for all values of x_2 superimposed, with $\psi_{12}(x_1, x_2)$'s phase as color and its magnitude squared along the vertical axis, and essentially provides a "side-view" of the top plot. The lowest plot is the (negative) charge density of $\psi_{12}(x_1, x_2)$, defined as

$$\rho(x) \equiv -e \int \left|\psi_{12}(x, x_2)\right|^2 dx_2 \tag{87}$$

Since these plots of $\tilde{\psi}_{12}(x_1, x_2, \upsilon)$ are the portions of the time evolution of ψ_{12} that oscillate with particular and distinct frequencies, we propose that these plots show the energy eigenstates of ψ_{12} in the position representation. The spatial structure of these wavefunctions are suggestively similar to two-particle products of the states seen in Figure 23. However, since their energy eigenvalues were seen not to be a simple sum of the lowest two eigenstates of the one-electron case, it seems plausible that these wavefunctions are not identical to simple products of the one-electron eigenstates.

Besides being inherently antisymmetric, these states' spatial structure becomes more complex as the energy is increased, lending increased plausibility for the proposal that they are energy eigenstates. This phenomenon is well-known in the structure of the three-dimensional atom. Further, in the eigenstates beyond the ground state, the charge density plots show peaks increasing with energy and flanking the central charge concentration. These extra peaks could provide a charge shielding effect, a phenomenon also known in the three-dimensional atom as "shell structure". So we also propose that these flanking peaks in the one-dimensional atom also show shell structure.

E. Conclusion

Using an arbitrary set of initial conditions, a great deal of the physics of the fermionic two-electron one-dimensional atom was determined. The simulation and analysis provided a prediction for a portion of the energy structure of this system, and, at the same time, provided detailed information on the spatial structure of the eigenstates that correspond to that energy spectrum. By choosing more energetic initial conditions, it is a easy matter to excite higher states of the system and determine the higher energy eigenvalues and eigenfunctions. The signal-to-noise ratio in both the energy eigenvalue and eigenstate data is also very strong, demonstrating the robustness, stability, and reliability of the code. The investigation also led to support for a hypothesis for shell structure in the one-dimensional atom, which had not before been observed. These findings show the applicability of code to obtaining results about arbitrary quantum systems.

VI. Energy Fluctuations in a Plasma

A. The Problem

This next problem has called for the largest simulations yet run using the quantum PIC code. Credit regarding computational resources goes to Jose Louis Hales-Garcia and Jan de Leeuw of the Department of Statistics at the University of California, Los Angeles, for their permission and generous contributions of computational time and assistance in using their 16 node Power Macintosh G4/400 cluster.[60] Several computational runs, taking up to two weeks of continuous processing time each, were completed on their cluster. This system was combined with software provided by the AppleSeed Project [36,61], making the computations possible.

Numerous properties of a classical plasma are well known. For example, a fundamental behavior of a plasma is the collective behavior of its particles at a

frequency known as the plasma frequency. In the case of an electron plasma, it is:

$$\omega_p^2 \equiv \frac{4\pi e^2 n}{m} \tag{88}$$

where m is the mass and n is the volume density of the plasma particles. Modes at this frequency are commonly detected in experiment, and these observations are easily predicted by considering perturbations in the collective motion of the particles due to their mutual Coulomb interactions.[62] In addition, a phenomenon known as Debye screening occurs, which has a characteristic length scale known as the Debye length:

$$\lambda_{Debye}^2 \equiv \frac{T}{4\pi e^2 n} \tag{89}$$

where T is the temperature in units of energy.

It is also well known that plasmas exhibit electromagnetic and density fluctuations in thermal equilibrium. Detailed studies of these electromagnetic fluctuations have been performed by Dawson[63,64], Rostoker et al[65], Sitenko et al[66], and Akheizer et al[67]. Most of these results were compiled in books by Sitenko and Akheizer et al.[68,69] A long tradition in plasma studies using the analysis of the fluctuations in a plasma exists because it is a powerful tool to investigate intrinsic properties of the plasma, such as its energy, screening effects, and diffusion. The spectra of the longitudinal fluctuations in the electric

field of an isotropic plasma [68,69], in the classical case, has long been taken to be:

$$\frac{1}{8\pi}\left|E_L(k,\omega)\right|^2 = \frac{T}{\omega}\frac{\mathrm{Im}\,\varepsilon_L}{\left|\varepsilon_L\right|^2} \tag{90}$$

where ε_L is the longitudinal permittivity of the plasma. For systems near thermal equilibrium, the fluctuation-dissipation theorem is useful because it provides an estimate of the energy in the electric field for all frequencies and wavenumbers. Therefore, one only needs to determine the dielectric permittivities to obtain a complete description of this diagnostic of a plasma at equilibrium. The calculation not only includes the energy in the fluctuations in the well defined modes of the plasma, such as plasmons (where $\varepsilon_L \approx 0$), but also the energy in the fluctuations that are not true propagating waves (where ε_L is far from zero), the so-called quasi-modes associated with the random motion of the particles. It is well known that for high wavenumber and frequency in classical plasmas at equilibrium, (90) follows a Gaussian as a function of frequency.

Our problem concerns such fluctuations in the electric field in a hot, dense electron plasma while accounting for effects due to quantum mechanics. In particular, we wish to consider an electron plasma in the parameter regime when the Debye length is similar to the de Broglie wavelength: $\lambda_{deBroglie} \equiv \hbar / p$. The question is: Are these fluctuations in such a plasma different from what would be expected in a completely classical model of that plasma? And: If so,

how are they different?

This question, inspired by a series of articles by Opher and Opher [70-74], has importance because of its relevance to stellar evolution. Stellar models are constructed by solving the basic stellar structure equations. The solution of these equations requires specifications of the opacity, nuclear reaction rates, and equation of state. In stellar evolution calculations for normal (non-compact) stars, the plasma is treated as a mixture of ideal gases.

However, previous astrophysical calculations about the plasma of the stellar interior assumed only classical mechanics applies. An electron plasma with the plasma parameters of many stellar interiors can reach conditions where the Debye length and de Broglie wavelength are similar. We should emphasize that Opher and Opher *did not* investigate a regime where particles are treated quantum-mechanically; they considered classical particles in quantum electromagnetic fields. However, to the extent they incorporated quantum mechanics in their plasma calculations, they predict that the resulting energy density is measurably different. They find that, for high densities, as in the interior of stars and in the early universe (e.g., for $T = 100$ eV and $n > 10^{24}\text{cm}^{-3}$, when the de Broglie wavelength is comparable to the interparticle distance), the assumption that the plasma behave purely classically is not valid.

If the answer to the above question is yes, that is, if plasmas do behave differently from what would be assumed using classical mechanics, then models

of stellar evolution must be revised, which then forces estimates regarding the ages of the stars and galaxies and the evolution of the early universe to be reconsidered.

The work in this chapter regards using the quantum PIC code to test this question. One of the authors of the work (M. Opher) that inspired this question has provided direct assistance with this work. No other code capable of modeling a plasma has as complete a model of quantum mechanics, and, because previous quantum codes [75] in this physical regime can handle only a few particles (~2), no other code that models quantum mechanics as completely can model as large a problem as the quantum PIC code can. These features make this work highly unique.

B. The Model and the Analysis

The plasma is modeled as a collection of electrons in a one-dimensional box mutually repulsed by their electrostatic (Coulomb) fields using the quantum PIC code. Note that this code models the electrons using quantum mechanics, while the fields are assumed to be classical. These properties of this model are distinct from those of the model in Opher's and Opher's published theory. Their theory uses a three-dimensional model of classical particles and studies fluctuations in an electromagnetic field modeled quantum-mechanically.

According to the predictions of Opher and Opher's theory, the difference that quantum mechanics makes is seen in the fluctuations due to the non-propagating quasi-modes in the electric field in the system. So it is of interest to see how different the fluctuation level of these quasi-modes are for a system of quantum particles. Therefore, we focused on studying the energy density of the electric field as a function of wavenumber k and angular frequency ω, which is the left side of (90). We believe studying this energy density on a plasma in the relevant parameter regime should provide evidence needed to answer our proposed question.

C. Implementation

The positions and momenta of the plasma particles are initialized in a way inspired by one of the earliest "sheet model" plasma simulations. [23] The simulation contains N quantum particles, each begun as a Gaussian of the form described by (65). The distribution of initial momenta p_0 is Maxwellian, like that of a plasma at equilibrium, at a given temperature. The distribution of initial positions x_0 are computed by adding a random Maxwellian distribution to the coordinates of regular lattice points in the simulation space. One lattice point is used for each quantum particle, and the variance of the distribution is equal to the lattice spacing. The standard deviation σ of the Gaussians are set

to four grid points. The random number generators of the original plasma PIC code were used to create the Maxwellian distributions. Listing 1 shows the loop that sets these initial conditions. Figure 25 depicts the particle placements.

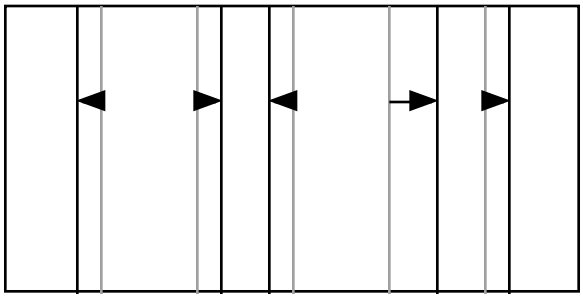


Figure 25. Placement of initial positions of the particles. The dashed lines indicate the lattice points, and the solid lines are the particle positions offset from the lattice points.

The code is run using these initial conditions and the infinite square well potential. The number density of the plasma is the number of quantum particle divided by the width of the well. The number of quantum particles ranged from 32 to 128, the well widths varied between 1016 and 2040 grid points, and the number of time steps was between 512 and 1024. The resulting quantum data sets were each on the order of 256 MB to over 1 GB in size. To ease the memory requirements of the quantum PIC code, adjustments to the virtual classical particle array allocation and the use of its indices throughout the code were made. These adjustments enabled the code to reuse the same virtual classical particle array for all quantum particles.

Since the electric fields calculated in the quantum PIC code subtract the

contribution due to the particular quantum particle being pushed (see Section C of Chapter III and (57)), the total electric field is not calculated explicitly during the simulation. Therefore, a separate code was derived from the quantum PIC code that reread the entire quantum data file and used the same field solvers to calculate $E(x,t)$, the electric field as a function of time and space, for the entirety of the run. Another code then performed a two dimensional FFT on this output to derive $E(k,\omega)$ and determine the energy density. Cross-sections of the energy density are then studied for comparison to the theory.

D. Proper Comparison

In theoretical predictions regarding plasmas, it is most often assumed that $n\lambda_{Debye}^3 >> 1$ (or, in one dimension, $n\lambda_{Debye} >> 1$). However, in a practical computational simulation, $n\lambda_{Debye}^3$ may be much lower than its value in the plasma one is trying to model. For proper comparison of theory and computation, we should bear in mind the consequences of representing a plasma using a smaller number of macroscopic (or finite-size) particles before making judgments about the predictions of quantum theory versus that of classical theory. Studies of the consequences of finite-sized particles for the fluctuation-dissipation theorem have been made by Langdon.[76]

Since the classical theory of plasmas and plasma codes based in classical

mechanics are at our disposal, we wanted to determine what kind of predictions would we see given only these tools. We wish to compare our diagnostics of the classical plasma code against the predictions of classical plasma theory, so we need to determine the dielectric permittivity of (90). In a one-dimensional classical plasma, the permittivity is calculated [62] using

$$\varepsilon_L(k,\omega) = 1 - \frac{\omega_p^{\,2}}{k^2}\int_{-\infty}^{\infty} \frac{1}{kv - \omega - i\eta}\, k\frac{\partial f(v)}{\partial v}\, dv \tag{91}$$

where $f(v)$ is the velocity distribution function of the plasma, and q is the charge of the particle. Inserting a Maxwellian distribution for $f(v)$ in (91) leads to

$$\varepsilon_L(k,\omega) = 1 + \frac{k_D^{\,2}}{k^2}\left\{1 - z\exp(-z^2/2)\int_0^z \exp(w^2/2)dw + iz\exp(-z^2/2)\sqrt{\frac{\pi}{2}}\right\} \tag{92}$$

where

$$z \equiv \frac{\omega}{k}\sqrt{\frac{m}{T}} \tag{93}$$

and

$$k_D^{\,2} \equiv \frac{4\pi q^2 n}{T} \tag{94}$$

A plot of (90) using (92), (93), and (94) yields Figure 26.

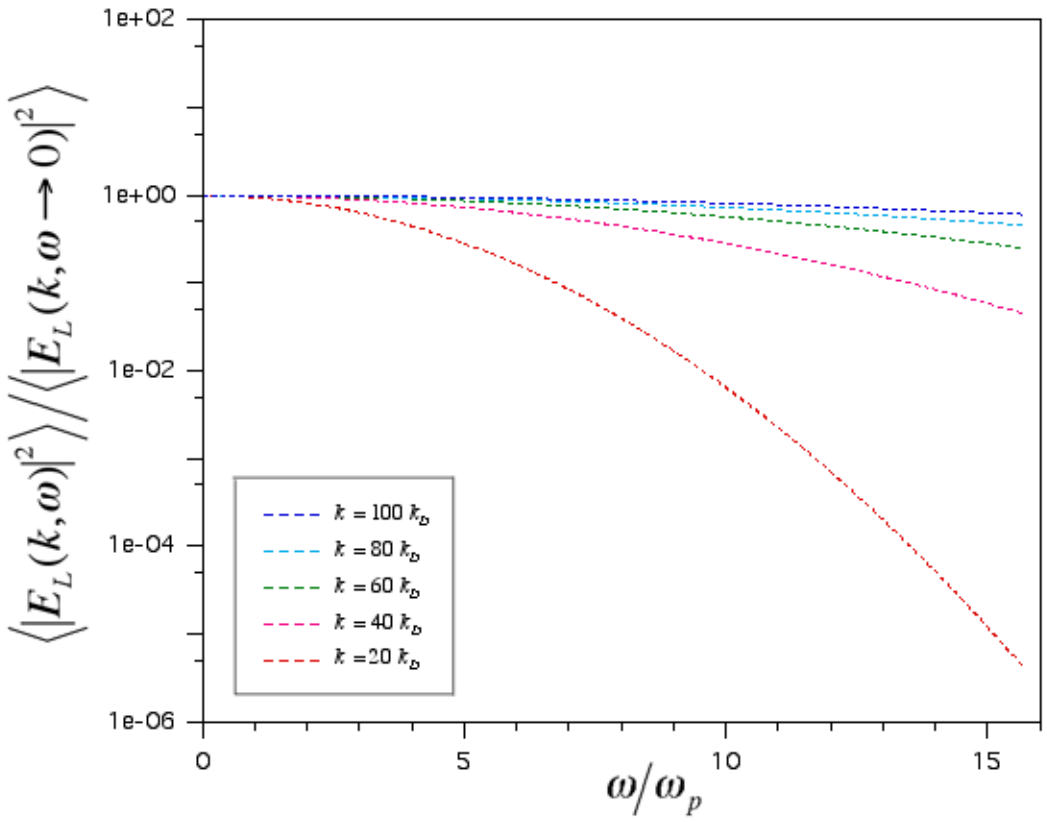


Figure 26. Energy spectra of the longitudinal electric field in a hot plasma assuming classical theory. The horizontal axis is frequency, and the colors of the plots indicate spatial wavenumber, with blue being the highest.

$\langle|E_L(k,\omega)|^2\rangle$ is a two-dimensional function, so we describe our convention to represent this energy density in one-dimensional plots here. In Figure 26, the energy density is plotted as a function of frequency in a semi-log plot. The colors indicate plots of different wavenumber, red being lowest and blue being highest. The red is when $k = 20\ k_D$, the magenta when $k = 40\ k_D$, the green when $k = 60\ k_D$, the cyan when $k = 80\ k_D$, and the blue when $k = 100\ k_D$. Each plot at a constant k is normalized to 1 at $\omega = 0$. For consistent comparison to the

quantum results, these plotting characteristics form the convention used for the remainder of the energy density plots in this chapter.

Figure 26 tells us that, in a purely classical model of a plasma, we would expect the energy density of the plasma to decrease with increasing frequency. However, with increasing wavenumber, the rate of the decrease as a function of frequency would decrease. These properties lead to a characteristic behavior of the normalized energy density of higher wavenumbers always being above those of lower wavenumbers. As the wavenumber approaches infinity, the energy density becomes constant as a function of frequency.

We performed a classical plasma simulation and extracted the energy density of the electric field in the same parameter space used for Figure 26. The simulation used a classical plasma PIC code (beps)[77] that led to the code that the quantum PIC code was originally based on.

A plot like that of Figure 26 using data from a run using 128 classical particles in a 1024 grid-point space in the classical plasma code is shown in Figure 27.

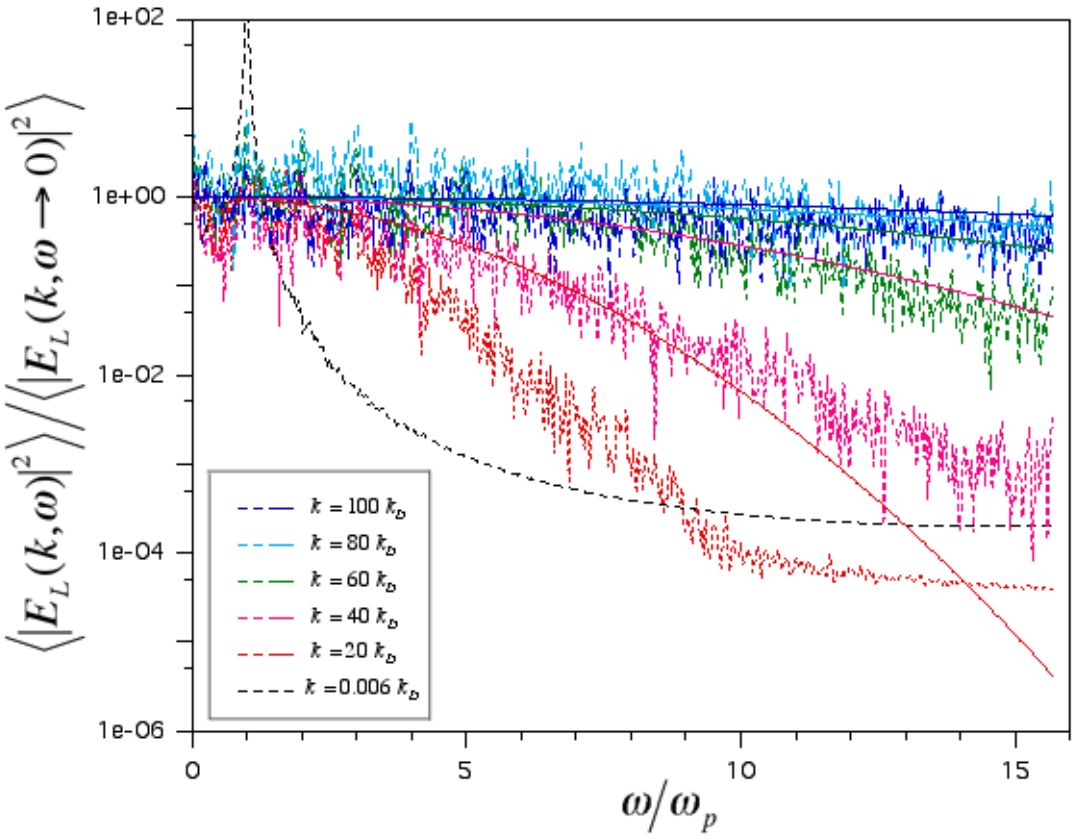


Figure 27. Energy spectra of the longitudinal electric field in a hot plasma using a 128-particle plasma simulation that assumes classical theory. The dashed curves are from the classical simulation, while the solid curves are from classical theory.

Figure 27 shows both solid curves for the theory and and dashed for the simulation. At first glance, the noise in the data obscures some of what the theory and simulation have in common, and some of the discrepancies seem rather large. We are encountering the consequences of low particle statistics, regardless of classical or quantum theory. Upon further study, Figure 27 does show that the classical simulation shows many of the same characteristics of the

theory, such as the increasing normalized energy density with increasing wavenumber, and the energy density becomes nearly constant with the highest wavenumbers. And we can clearly see, at low k, a resonance at the plasma frequency, confirming a fundamental collective effect in plasmas. (The $k = 0$ theory curve based on (90), (92), (93), and (94) was off scale.) Before making further judgment, one should study what happens when the number of particles in the plasma simulation is significantly increased.

Figure 28 shows results from a simulation like that used for Figure 27, but with one hundred times as many particles. The macroscopic plasma parameters were the same.

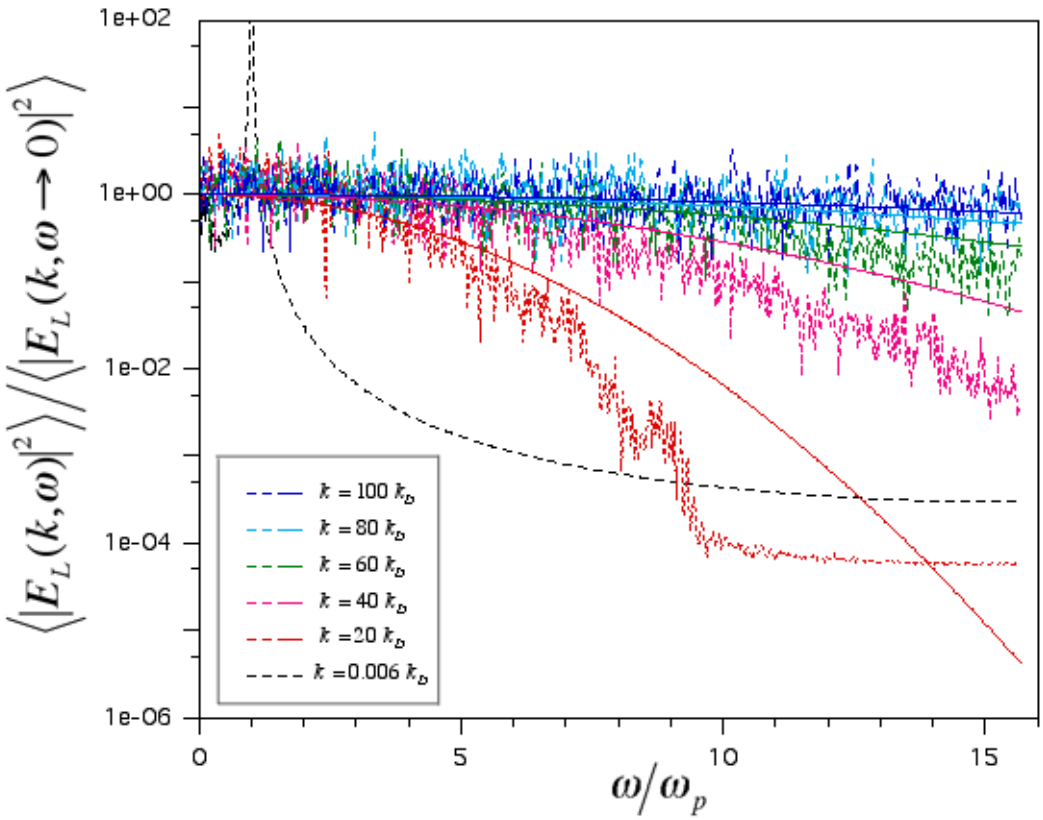


Figure 28. Energy spectra of the longitudinal electric field in a hot plasma using a 12,800-particle plasma simulation that assumes classical theory. The solid lines are theory, while the dashed is the simulation.

With a significant increase in the particle number, we can see that the simulation is behaving more like the classical theory. Except for a portion of the $k = 0$ case, the monotonically increasing energy with wavenumber is preserved, with the constant energy density as wavenumber approaches infinity. However, the increased number of particles allows the curves to more closely follow the theoretical predictions. Note that, in the red, $k = 20\, k_D$, curve the simulation falls below the theory, then at some point it becomes smooth and constant. Also present in the $k = 0$ case, this behavior may indicate a noise floor present in this diagnostic of the simulation.

Further, we were able to produce a classical simulation with almost another hundred times as many particles. These results are shown in Figure 29.

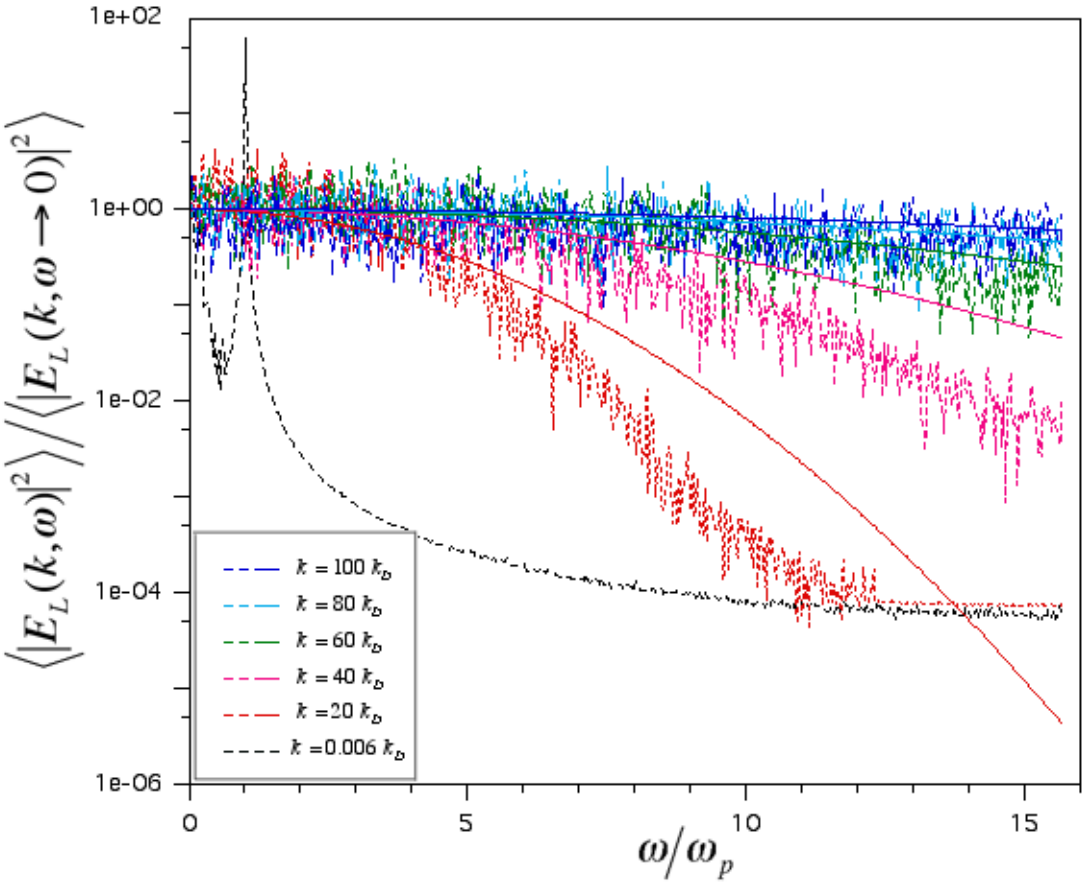


Figure 29. Energy spectra of the longitudinal electric field in a hot plasma using a 1,024,000-particle plasma simulation that assumes classical theory. Solid lines are theory, and dashed lines are from the simulation.

Note that the characteristics of these plots continue to be preserved, while the noise level has decreased after increasing the particle number. The resonance at the plasma frequency at low k is well resolved. Also, a significant difference between Figure 28 and 29 is that the noise floor is encountered at a higher frequency, as can be seen in how the red graph better follows the characteristics of the theory before becoming constant. Further, this floor matches that of the

$k = 0$ case.

With this study of the consequences of finite particle count in mind, we may look upon the 128-particle case shown in Figure 27 in proper perspective. Based on the difference in noise floor observed in Figures 28 and 29, we are inclined to extrapolate that the noise floor should have a larger effect in the 128-particle case. Also, we would expect the noise level on the "signal" we wish to deduce from Figure 27 to be significantly higher than that seen in Figure 28 and 29. From this study, it is evident that there are characteristics of this diagnostic due solely to low particle statistics. To the extent of the commonality of the studies, these results are consistent with the study by Langdon.[76] These classical simulations are using PIC (or finite size) particles. To reproduce the theoretical curves, we would need to both increase the number of particles *and* decrease the particle (cell) size compared to λ_{Debye}. (An alternative approach might be to reinterpret the theoretical results by incorporating into (91) what we would expect due to the shape function of these PIC particles.) These phenomena should be kept in mind before making judgments about the results of other simulations using similar particle numbers.

E. Quantum Theory and Simulation

Next, we wish to consider predictions that incorporate quantum

mechanics. The theories presented by Opher and Opher assumed three-dimensional classical ($\lambda_{deBroglie} << \lambda_{Debye}$) particles interacting with a quantum-mechanically modeled electromagnetic field. This quantum PIC code is one dimensional and assumes a classical electrostatic field. Therefore, for a proper comparison, a new set of theoretical predictions must be made that are relevant to this experiment.

(90) is, in fact, an approximation.[68,69] It is based on taking the limit as $h \to 0$ of

$$\frac{1}{8\pi}\left|E_L(k,\omega)\right|^2 = \frac{\hbar}{\exp(\hbar\omega/T)-1}\frac{\operatorname{Im}\varepsilon_L}{\left|\varepsilon_L\right|^2} \tag{95}$$

Therefore, our prediction of the plasma model incorporating quantum theory uses (95) without assuming that $h \to 0$ while using the same dielectric permittivity described by (92), (93), and (94). We base this approach on the following. In Opher's and Opher's theory, one of their first proposals is not to assume that $h \to 0$. Then, they assume a calculation of a quantum-mechanical electromagnetic field in their calculation of the dielectric permittivity. Since the field in our simulation is classical, we, for the purposes of this comparison, assume a dielectric permittivity as described by (91). We also wish to assume a Maxwellian velocity distribution for the plasma of quantum particles so that we can directly compare against the classical plasma result. We should note that, although we have techniques to handle antisymmetric wavefunctions in

principle, the computational requirements of such a procedure for 128 particles make the task impractical. Thus, we are not incorporating electron degeneracy effects into this model.

Using this model of the plasma possessing the same macroscopic plasma parameters used for Figure 26, we are able to compare the quantum theoretical prediction with the classical theoretical prediction. Also, in this case, the mean de Broglie wavelength (specifically, the de Broglie wavelength for an electron moving at the thermal velocity) was approximately twice that of the Debye length. These predictions are compared in Figure 30.

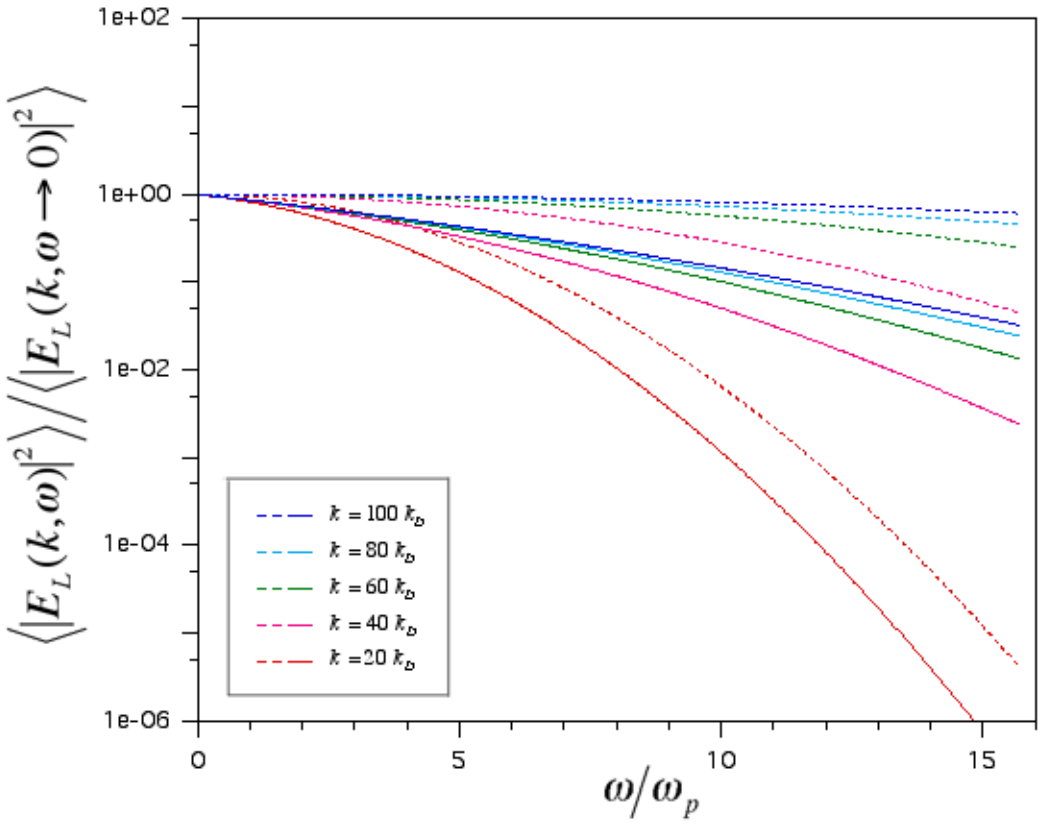


Figure 30. Energy spectra of the longitudinal electric field in a hot plasma using quantum theory and classical theory. The solid curves is from the quantum

prediction, while the dashed ones are due to the classical prediction.

The monotonically increasing normalized energy density with increasing wavenumber is present in both the quantum and classical theory. But the key difference seen in the quantum theory is that, at the same wavenumber and frequency, the normalized energy density is *below* that of the classical theory. This behavior can be seen in the red, $k = 20\, k_D$, curve. In fact, we can see that the discrepancy between the classical prediction and quantum prediction becomes more pronounced as wavenumber increases. This discrepancy is a key characteristic that we wish to see in results from the quantum simulation.

For comparison to the quantum simulation, we created a plasma in the quantum simulation possessing the macroscopic plasma parameters used for the preceding figures in this chapter. Frames of a run using the quantum PIC code to simulate 128 particles in a 1016 grid-point space are shown in Figure 31.

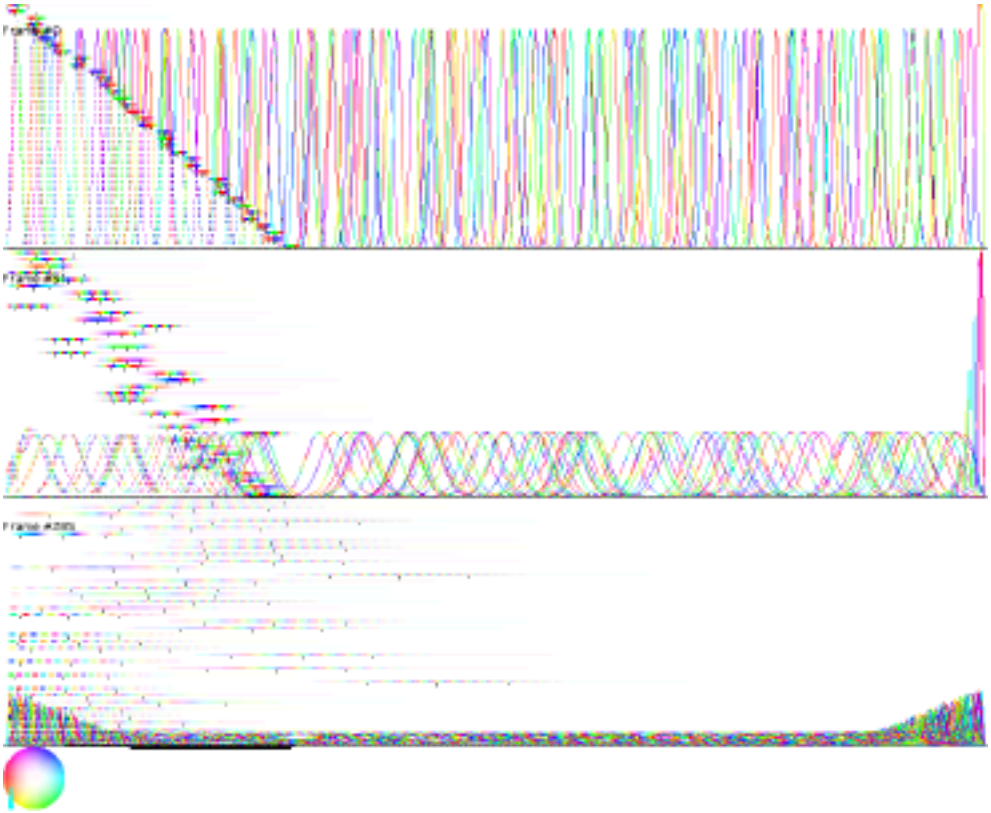


Figure 31. Three frames from the evolution of a set of Gaussians representing a hot plasma.

We should reiterate that an important characteristic of this electron plasma is that the mean de Broglie wavelength was approximately twice that of the Debye length. In this regime, it becomes plausible that plasma effects compete with those of quantum mechanics. This ratio fixes an expression relating well-known plasma parameters:

$$T = \hbar\omega_p \left(\frac{\lambda_{Debye}}{\lambda_{deBroglie}} \right) \tag{96}$$

where the momentum of an electron moving at the thermal velocity is used for

the de Broglie wavelength. Assuming the above length ratio and a density of $10^{27}\,\mathrm{cm}^{-3}$ gives a temperature of 4.2 keV, or about $4.8\times10^{7}\,\mathrm{K}$. According to the literature [62], these parameters are consistent with the conditions of a stellar interior.

This simulation took approximately two weeks to complete on a cluster 16 G4/400's. Plots of the simulation's electrostatic energy density are shown in Figures 32 and 33.

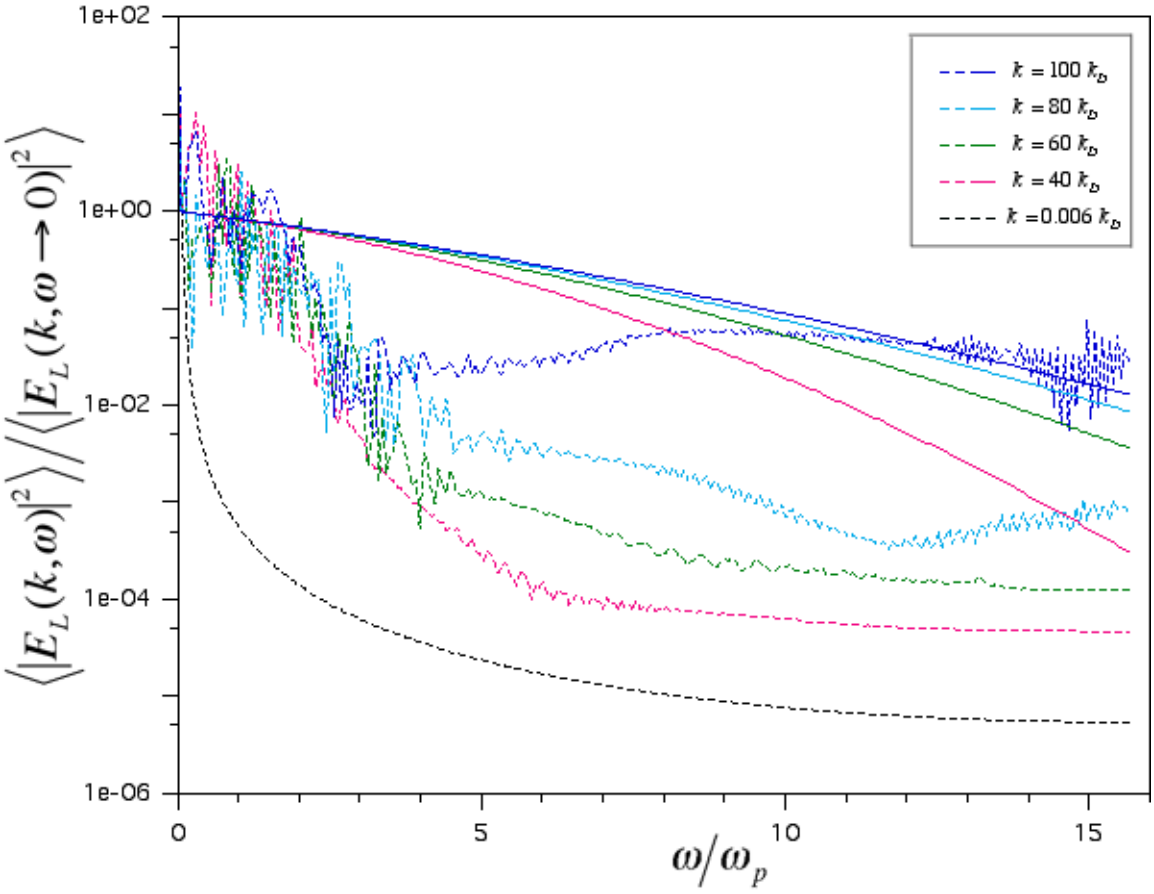


Figure 32. Energy spectra of the longitudinal electric field in a hot plasma, shown in Figure 31, while including quantum-mechanical effects. The horizontal axis is frequency, and the color of the plot indicate spatial wavenumber, with blue being the highest. Results from the 128-particle quantum simulation are shown in dashed curves, while the quantum theoretical prediction (seen in Figure 30) is shown in solid curves.

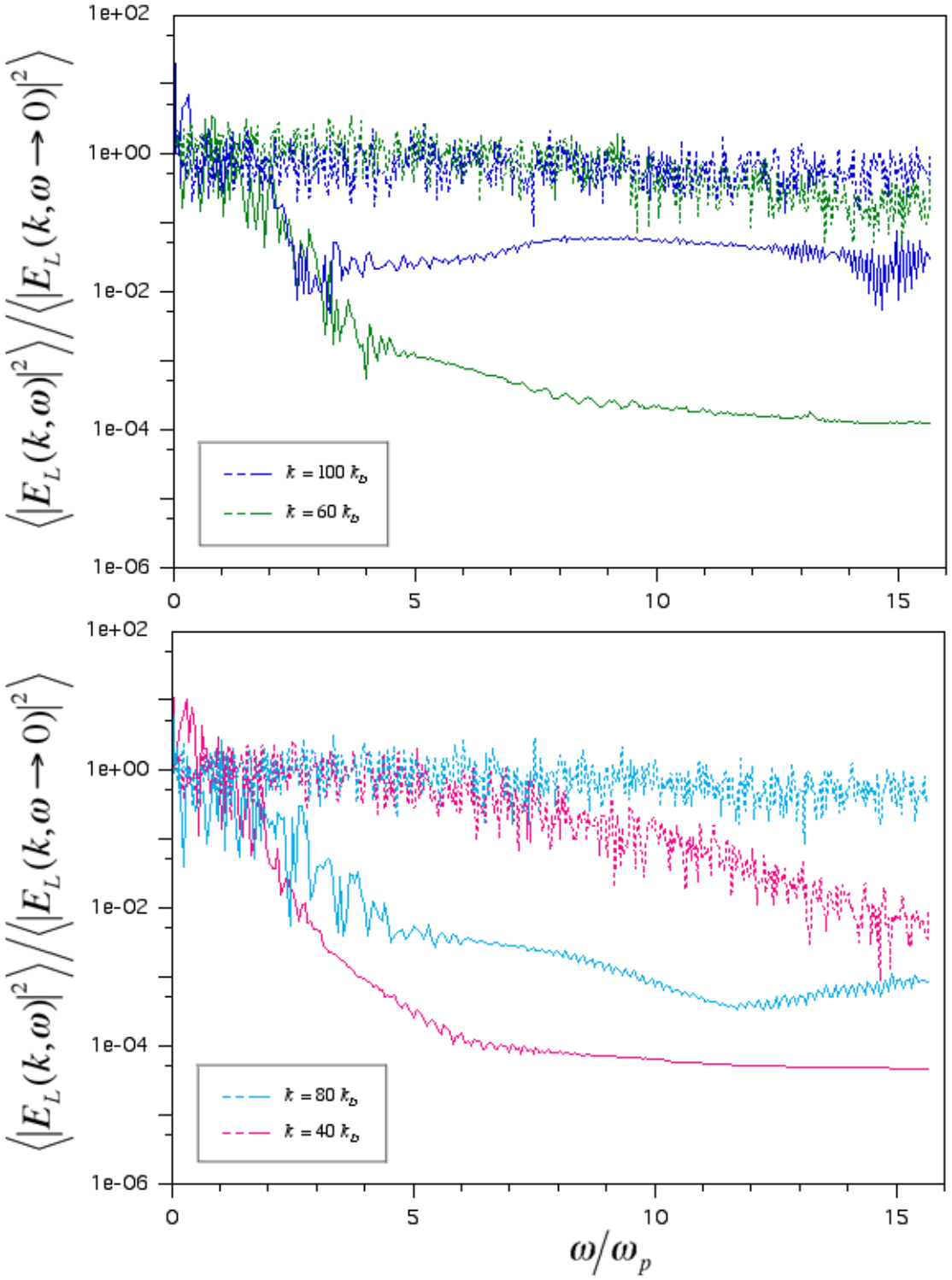


Figure 33. A comparison against a classical simulation of the energy spectra of the longitudinal electric field in a hot plasma, shown in Figure 31, while

accounting for quantum-mechanical effects. The horizontal axis is frequency, and the color of the plot indicate spatial wavenumber, with blue being the highest. Results from the 128-particle quantum simulation are shown in solid curves, while results from the 1,024,000-particle classical simulation is shown in dashed curves. The plots were split into two to make the curves easier to distinguish.

The rise in the plateau at high frequency as a function of wavenumber indicate the non-propagating quasi-modes. As in the earlier results, the quantum simulation shows increasing normalized energy density with increasing wavenumber, as we would expect from the quantum theory shown in Figure 30. Figure 32 also shows some of the effects visible in Figures 27 and 28 that are characteristic of low particle number, such as how the computational results dip below the theoretical results then flatten.

In Figure 33, the quantum results are shown in solid curves, while the results from the 1,024,000-particle classical plasma simulation are shown in dashed. We show these results to compare the consequences of using quantum mechanics as opposed to classical mechanics. It was fortunate that this 128-particle quantum simulation possessed a low enough noise level for comparison.

By comparing the normalized energy density from the two simulations at the same wavenumber, we can see a distinctive and consistent discrepancy between the quantum prediction and the classical prediction. For example, in the highest wavenumber (blue) plots, the classical simulation shows an almost

constant energy density as a function of frequency. However, the quantum simulation shows an energy density that clearly decreases and stays below its value at zero frequency. Based on the study of low particle statistics made in the last section, it should be reasonable to believe that the decrease in energy density with frequency is arrested by a noise floor similar to that observed in Figures 27, 28, and 29. The character of the noise floor can be seen in all of the plots of the quantum data.

However, even with the limitations due to low particle number, the normalized energy density from the quantum simulation remains measurably and consistently *below* that of the classical simulation at the same wavenumber. This discrepancy is much like what we observed from the theoretical predictions shown in Figure 30. This observation lends credence to the proposal that quantum effects can have a measurable effect in plasmas which were formerly treated as “classical plasmas”.

F. Conclusion

The results of this analysis are very promising. Not only do the plots in Figure 33 contain many of the qualitative characteristics predicted by a theory that includes certain quantum effects, but we were able to show a measurable difference in the predictions of the quantum code compared to that of a code

based solely on classical mechanics. The similarities between the quantum PIC code's output and the quantum theory, embodied in Figures 30, 32, and 33, indicate the code is duplicating at least some of the physics involved and has significant bearing on the question at hand. With a more rigorous quantitative analysis using plasma and quantum theory as it applies to the circumstances modeled in the code, these simulations should carry weight in confirming the correctness of proposals incorporating quantum mechanics into plasma models.

Further study is needed, not only on the quasi-modes seen in this chapter, but on nonadiabatic modes (including those close to the plasma frequency) of the plasma. In fact, a portion of the data from the same quantum PIC simulation shown earlier reveals an intriguing effect. In a classical plasma, a characteristic effect is a resonance at the plasma frequency, which can be easily seen in the black curves of Figures 27, 28, and 29. Data from the quantum simulation at low k is shown in Figure 34.

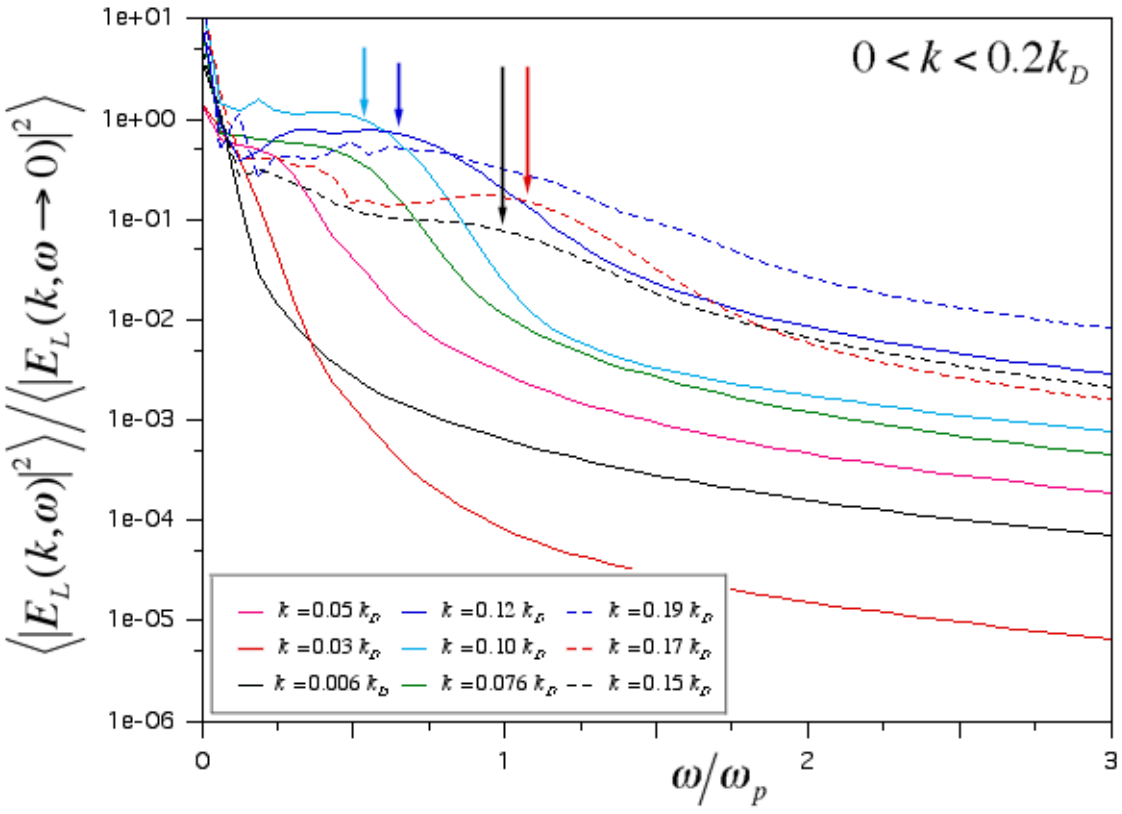


Figure 34. Energy spectra of the longitudinal electric field in a hot plasma at low k while accounting for quantum-mechanical effects. The arrows point out rises in the energy density, some of which could be similar in origin to the peak at the plasma frequency seen in the classical results.

If this were a classical plasma, a sharp rise at or near the plasma frequency would be expected in the plots with the lowest values of k. However, in its place, we see a small rise in the quantum data. Since the de Broglie wavelength in this simulation was greater than the Debye length, perhaps quantum effects have largely smothered the plasma effects. However, these simulations indicate that well-defined collective plasma modes of quantum particles (as

opposed to ballistic modes, a likely cause of the behavior seen in Figures 32 and 33) are still present. The density of the quantum particles was low (average quantum particle density times the Debye length was less than one) by standards of a normal plasma, so some plasma effects could be obscured by the low particle statistics. Clearly, these issues are deserving of future work.

The work shown in this chapter logically leads to further steps. Methods to create closer comparisons of theory and simulation are needed. Because of the well-known characteristics of the classical code, it would be wise to experiment with techniques such as smoothing or averaging (e.g., over wavenumber and/or frequency) of data from the classical code. In addition, a study of the origin of the noise floor, seen in plots from both the classical and quantum codes, would be appropriate. This study applied to the classical simulations could lead to a prediction of the noise floor relevant for interpretation of the quantum data. Work by Langdon [76], suggests that similar types of noise may be reduced by decreasing the time step or increasing the duration of the simulation. Although it would obviously require more CPU time, it is otherwise a simple experiment to try. With these considerations, more direct comparisons between theory and simulation would become easier to perform.

Further, approaches, alternative to that of (88), to estimating the electrostatic energy density can be considered. One may derive the electric field

using the permittivity as a dielectric response function of the density fluctuations in the plasma. This idea is inspired in a paper by Dawson. [78] For example, a non-interacting model of the plasma could be interpreted as an external charge density as seen in Ichimaru. [62] Some work on predictions using the density fluctuations was performed, but the model, so far, did not provide a comparison as close as the one presented here. Further work would be appropriate.

Regarding the low particle statistics, quantum simulations of higher particle number can be performed as well, with a corresponding increase in computation time, of course. Some of the recommendations for more efficient quantum simulation, presented in Chapter VII, can be performed for such simulations. Together, the proposals described above paragraphs would take, at minimum, six months of further work.

Further in the future, it may be possible to use this code to predict reaction rates in a plasma. Estimates of nuclear reaction rates in some extraterrestrial plasmas are higher than expectations inferred from Earth-based laboratory experiments. In a plasma, ions are not “naked”, but are surrounded by electrons that form a shielding cloud around them. The polarization clouds partially screen their charges resulting in a lower Coulomb barrier between them, thus providing an enhanced tunneling probability. In addition, collisions are occurring, not just between particles and fixed shielding clouds, but

between particles and fluctuating shielding clouds. Consequently, the potential barrier, and therefore the penetration rate, fluctuates. These phenomena should be taken into account. For these purposes it may be desirable to create a two-dimensional version of the code and include ions.

VII. Future Work

A. The Future

The quantum PIC code has a great deal of potential utility, demonstrated by the examples shown in this dissertation. In addition, it is possible to evolve the code into a new tool for the future. This code and the experience accumulated in building it may serve as a guide for codes not yet written that explore new possibilities. It is the hope of this author that this work is important, not simply for itself in isolation, but for the future development to which it leads. This chapter is meant to serve as a collection of suggestions for such future work.

B. One Dimension

Leveraging off of Chapter V, further analyses of the one-dimensional atom may be pursued. The two-electron studies of higher energy eigenstates may be extracted from further runs of the type described in that chapter, and superposition descriptions of those eigenstates in terms of the one-electron eigenstates could be determined. Also, more electrons could be added to analyze states of higher Hilbert spaces. The calculations involved to construct and analyze an antisymmetrized state of these higher dimensions will be more involved and are likely to grow in computational cost exponentially as a function of quantum particle number. Visualization of the three-electron case could utilize volumetric raytracing techniques developed for other visualization software [79]. This visualization could then be used to slice the higher-dimensional states that represent more electrons. Practical considerations, such as storage space, may limit these studies to roughly six electrons.

It may be possible to apply techniques developed for a pair of antisymmetrized electrons to construct simplified models of multielectron states. Usually, the behavior of electrons with probability densities that are far apart are affected little by their antisymmetric properties, but those that approach each other generally are affected. Perhaps an approximation of the fully antisymmetrized N-electron state could be constructed by

antisymmetrizing only each closest pair of electrons (determined by considering properties of their individual probability densities, such as their overlap) while calculating a simplified version for those that are far apart. This technique may serve useful for the analysis or for the simulation itself, e.g., if calculations based on the N-electron wavefunction are desired to influence the simulation directly. The infinite square well, the simple harmonic oscillator, and the one-dimensional atom may serve as testing environments for such experiments with antisymmetrized states.

Using Chapter VI as a starting point, it is possible to pursue further comparisons to the proposal about quantum effects in a plasma. Additional tests with other plasma parameter regimes may serve to provide useful comparisons with the theory. In addition, one may add nuclei to the quantum PIC code and use the simulation to estimate nuclear reaction rates by observing how these nuclei behave. Because of the considerably greater mass of the nuclei, their de Broglie wavelengths will be much smaller, so it is likely to be sufficient to model these nuclei as classical particles, not unlike the original plasma code. This hybrid plasma PIC / quantum PIC code could contain code that models a number of classically modeled positively charged protons or deuterons interacting with an equal number of quantum-mechanically modeled negatively charged electrons. Since there would be so few nuclei by comparison to the number of virtual classical particles, the additional

computation cost should be immeasurable, and the additional plasma PIC pieces should weave easily with the existing quantum PIC code. This author recommends beginning with existing quantum PIC code, then adding the relevant pieces to model the nuclei. The computational costs would require many days on a parallel computers with a few nodes and hours on a parallel computer with hundreds of nodes.

Additional tests of the semiclassical techniques used in the quantum PIC code can be explored, such as more accurate modeling of finite quantum barriers and wells. In the case of the rectangular barrier, rather than attempting to smooth the barrier edges, one could borrow the technique used for the boundary conditions of the infinite square well. In the external potential routine, the correct, stair-stepped potential may be added to the potential array, while nothing is added to the force array. Then, in the particle pusher, if a virtual classical particle is about to cross a barrier edge, a test on its momentum is made. Those virtual classical particles that have sufficient momenta to "climb" the potential may pass and suffer a momentum loss, while those that do not have the required momentum are bounced. Those virtual classical particles that begin inside the barrier and fall off are given a momentum kick upon exiting the barrier. Note that these momentum adjustments would occur sharply at the barrier edges, much like the infinite square well boundary conditions. In this author's opinion, this method should be sufficient to allow

for accurate modeling of quantum tunneling phenomena. Finite square wells would be modeled similarly, with the momentum behavior switched. This proposed method was conceived and considered while investigating the rectangular quantum barrier quantitatively, but was not attempted because its applicability seemed limited to that problem, and other problems in the course of this research had higher priority.

Another type of quantum system that can be straightforwardly addressed is those involving the Morse potential. It would be a matter of entering its form into the external potential routine and setting up initial conditions of interest. Evidence of other work [11,13,58] using this potential indicate it may be of significant interest in quantum and molecular chemistry.

What is convenient about addressing problems like these with this code is that these one-particle runs take only on the order of hours on modern personal computing hardware. This would allow progress to be made using limited computing resources.

C. Higher Dimensions

Like has been done in the past with plasma codes, this quantum PIC code may be extrapolated to higher dimensions. Study of the derivation, shown in Chapter II, was made for two- and three-dimensional cases. For a

great deal of the derivation, the replacement of momentum and space coordinates with their vector counterparts is straightforward (e.g., add vector signs and convert many multiplications to dot-products). For example, (55) would become:

$$\langle \mathbf{x}_f | \psi(t+\Delta t) \rangle = \sum_{\mathbf{p}_0} \sum_{\mathbf{x}_0} \exp(\frac{i\mathbf{x}_f \cdot \mathbf{p}_{cl\,f}}{\hbar}) \exp(-\frac{i\mathbf{x}_{cl\,N} \cdot \mathbf{p}_{cl\,f}}{\hbar}) \frac{\exp(iS_{cl}/\hbar)}{h^3\sqrt{\det(M)}} \langle \mathbf{x}_0 | \psi(t) \rangle (\Delta x \Delta p)^3$$

(97)

Likewise, the classical path iterative method and the definitions of quantities such as the Lagrangian and the action are easily extrapolated. With these extrapolations, the addition of electromagnetism fits better into this context (including the use of the canonical momentum $\vec{p} - \frac{e}{c}\vec{A}$). Also, with magnetism, the concept of spin can be incorporated, perhaps using a spinor representation to describe the evolution the wavefunction. Spin terms of the Hamiltonian are easily included in the effective potential. These spinor wavefunctions may require the modeling of a pair of virtual classical particles for every one that did not consider spin.

A point of greater potential difficulty is evaluating the determinant in this case. Consider the description given in Sections E of the semiclassical matrix and F of its determinant of Chapter II. While the matrix became tridiagonal in the one-dimensional case, it has been determined that the

corresponding determinant in the two- and three-dimensional cases behave as a product of determinants of triagonal matrices (one for each dimension) with the addition of terms, proportional to cross derivatives (e.g., $\frac{\partial^2 V}{\partial x \partial y}$) of the effective potential, that link matrix elements describing terms from different dimensions. The two-dimensional case was investigated sufficiently to determine that it should be possible to compute the determinant of the two-dimensional semiclassical matrix using a parallel pair of iterative methods. These iteration schemes are equivalent to a pair of finite difference equations possessing cross terms that link these equations together multiplied by coefficients proportional to the cross derivatives of the effective potential. Based on the analysis in two-dimensions, it seems reasonable to expect that a similar form would occur in three-dimensions.

However, a question should be considered: how important are these cross terms? Perhaps it is sufficient to follow the evolution of these finite difference equations, one per dimension, and ignore the cross terms. It has been observed in the one-dimensional quantum PIC code that the determinant evaluations result in multipliers close to one. Since these cross terms would be a correction to an already small effect, it may be sufficient to evaluate the determinant by considering the multidimensional semiclassical matrix independently by dimension.

Assuming the details of the semiclassical methods used for a multidimensional quantum PIC code can be resolved, such a code could provide great utility. A two-dimensional version could simulate the physical situations, named "quantum billiards", in the article by Heller and Tomsovic [13] instead with multiple quantum particles. The quantum corrals built with scanning tunneling microscopes could be modeled using any number of electrons. Such a code would also have great relevance to any model of phenomenon on silicon substrates, such as quantum dots and circuitry. The problem could be defined as electrons exploring a space with potentials determined by the circuit design. Considering the foreseeable end of Moore's Law and its consequences to computational hardware in the information age, quantum effects on the design of smaller and smaller transistors and other gate logic are likely to become significant. (Some of these questions may be answerable using the current code.) A two-dimensional quantum PIC code could help model and predict the behavior of proposed designs, perhaps extending the lifetime of the processor improvement rate the computer industry has so far enjoyed.

A three-dimensional quantum PIC code could have applicability to even more difficult quantum problems. In line with the above problem, three-dimensional potentials could be experimented with, adding a new dimension in circuit design, or provide understanding of other phenomenon such as electron

gases in metals. The simplest problem of a more fundamental nature would be multielectron atoms, duplicating shell structure and other phenomenon we know to occur in atoms. Early versions will probably model the nucleus classically, perhaps extending it to a quantum model later. With the atomic simulations established, then multiple atoms can be combined to form molecules, allowing us to analyze their behavior, including their internal oscillations. It may be sufficient to model such atoms with a few quantum electrons each. These electrons provide bonding while surrounding noble element based cores. Ultimately, hundreds of atoms could be assembled to form complex molecules to answer questions about complicated quantum problems such as protein structure and how they fold.

Combing electromagnetism with the three-dimensional quantum PIC code would add a wide space of problems. The simplest would be absorption and emission of light by single atoms. It may be sufficient to provide a classical model of electromagnetism for this purpose, much like how current electromagnetic plasma PIC codes operate. Similar phenomena can be modeled for molecules. A more radical application would be photodisassociation of atoms from molecules. Then, considering how such processes can absorb or emit significant amounts of light, chemical reactions could be modeled. Eventually, more accurate models may wish to extrapolate the semiclassical methods to relativistic [80] paths, since relativistic effects have

been observed in atoms.

In any case, this direction of exploration of quantum modeling methods is vast.

D. Evolution of the Implementation

Since it would be desirable to reduce the computation time necessary for simulations using the current code or future similar codes, we explore alternative computational methods to implement this type of simulation.

A modification to the current parallelization scheme could be attempted with the current quantum PIC code. Currently, each quantum particle is considered one at a time. Each quantum particle's virtual classical paths are evaluated, then that quantum wavefunction is reconstructed, and then the code moves on to the next quantum particle. It is possible to instead evaluate all the virtual classical particles of all the quantum particles at once. This scheme would require adding a quantum particle index to the virtual particle array and potential and field arrays, increasing their allocation size significantly, and appropriate adjustments to the code for virtual classical particles accessing information relevant to their corresponding quantum particles must be made. While the disadvantage is increased memory requirements, the advantage would be that the code would more efficiently utilize its particle manager

routine on a parallel computer. Since there would be more virtual classical particle information passed between processors per node, the message sizes would be larger, pushing the communications bandwidth usage to a more efficient regime. The effect would be more efficient parallelism during the particle push / particle manager loop, which is typically the most time-consuming portion of the code. (Depending on the problem and the number of processors, the distribution of CPU time is: 50-80% on the particle push / particle manager loop, 15-40% on the wavefunction reconstruction, and <10% on the field solve.) This scheme would show the greatest improvement on parallel systems with large (>100) numbers of nodes (which typically has ample amounts of memory).

Some of the routines can be accelerated using specialized hardware. As the plasma code has been vectorized in years past for vector processors such as those by Cray, the quantum PIC code can be vectorized for current vector hardware, such as the AltiVec instruction unit in the Motorola MPC74x0 PowerPC Microprocessor (a.k.a., the "G4" series). The particle pusher, since it is largely unchanged from the plasma code, has clear methods of vectorization just like how the plasma code's pusher was vectorized.

In addition, the wavefunction reconstruction routine can be vectorized. Consider the description in Section C of Chapter III, in particular, (61), (62), and Listing 5. After rearranging the arrays (such as `wtemp`) which hold the

accumulated virtual classical particle contributions so that space is the least significant index and it is aligned in memory on a vector boundary, the virtual classical particle contribution deposit may be rewritten in the following way. Let us assume the vectors contain n floating-point elements. (We provide this discussion with the AltiVec instruction set in mind, which has properties distinct from other architectures, such as those of Cray.) Promote `ctemp` and `cincr` in Listing 5 to `vector complex`, but alter their initialization. Load the elements of `ctemp` sequentially with $\{1, m, m^2, m^3, \ldots, m^{n-1}\}$ and multiply all the elements of `ctemp` by y_0. In the meantime, load all the elements of `cincr` with m^n. Then loop over the number of complex vectors that are in `wtemp` adding `ctemp` to the first vector, then multiplying `cincr` by `ctemp`, and continue the loop on the rest of `wtemp`. This completes a virtual classical particle deposit evaluation. The sum across processors at the end of this routine is easily vectorized as well. Of course, these are complex calculations, so such complex computations will have to be built in languages without complex intrinsics. Assuming the number of grid points in each processor is large compared to n (4 in the case of AltiVec), this vectorized code should provide a speed-up factor of almost n for the wavefunction reconstruction routine. A similar calculation for two- and three-dimensional quantum PIC codes should speed up at least as well as in the one-dimensional code.

In many simulations using the quantum PIC code shown in this

dissertation, each quantum wavefunction is somewhat localized in space. Another optimization method may be used that takes advantage of this observation. A virtual classical particle's contribution is proportional to the wavefunction at that particle's starting position (see Sections A and C of Chapter III, (55), and (59)). If the wavefunction magnitude at that position is small, then the computations on paths that start there are not as significant to the answer as others. It may be possible to recognize portions of the wavefunctions that are small (prioritized by $|\psi(x)|^2$, for example) and reduce the number of virtual classical particles that begin there, while proportionally increasing their contribution. This reduction in number and increase in contribution can be performed in stages based on $|\psi(x)|^2$. The momentum spread from that point could be reduced, made more sparse, or both. This technique would selectively reduce the sampling of phase space and reduce the computational cost to push one quantum particle. At best, this approach could reduce the number of needed virtual classical particles per quantum particle to be proportional to the volume of the space, rather than the phase space, of the simulation. One must be careful not to reduce too much, however, or else the delicate balance of cancellation due to phase could be disturbed. The author recommends one-particle tests like those shown in Chapter IV be performed to validate the code.

Assuming this technique is successfully implemented, the code would

then generate numerous classical paths in concentrated locations where the wavefunction probability density was most significant. In a PIC code on a parallel system, these concentrations lead to load imbalances: certain processors would be assigned more work than others. This imbalance could force other processors to wait for the overloaded processors to catch up. The solution would be to implement the alternative parallelization scheme described earlier in this section. Since the quantum particles, if they are localized, are most likely to distribute their localizations across many cells, then the virtual classical particle concentrations are likely to be well distributed among processors, so the load should become well-balanced once again. Such techniques can be combined with ones established for plasma PIC codes. [55, 81, 82]

E. Evolution of the Methods

Significant modifications to the semiclassical methods and their application could be made, for the purposes of greater computational efficiency, greater physical accuracy, or exploring new types of simulations.

There exist ways to reduce the cost of the numerical methods to push the virtual classical particles. Since the fields are held constant for many classical time steps, a multi-step method could be implemented, with the caveat of guaranteeing that the method can account for the partitioning of the fields

according to PIC methods. Another approach could be attempted to use few classical paths. Contributions of paths "in between" the calculated paths could be interpolated. Also, since most paths start out very close to each other, the number of calculated paths could start with only a few, then accumulate, generated from the interpolated contributions as the classical time steps exhaust the quantum time step.

Other approaches exist to reduce the number of virtual classical particles per quantum particle. Much of the virtual classical particle contributions cancel each other, typically in the range of high momentum. By looking at the grid points of the wavefunction near the starting point of the path, an estimate of the "primary" momentum of the wavefunction could be made. Virtual classical particles could be launched from that starting point in a narrower "momentum cone" around that primary momentum. Since action, for high momentum, becomes dominated by the kinetic term of the Lagrangian, contributions beyond that momentum cone could be approximated with an analytical solution. Such an analytical solution would have similarities to the Fresnel integrals. Because of the form of the kernel of such integrals, this method may become analytically simpler in the two-dimensional case (or in two-dimensional slices of the three-dimensional case) than in the one-dimensional case, as has been seen in other studies involving such integrals [83].

Another possibility is the use of wavelet theory. [84] Wavelets are a kind

of basis set that spans a phase space differently than a pure space or Fourier space representation. Wavelets have been used to identify frequency ranges of signals in a limited time interval. Since the phase space explored here is position space and its Fourier space, momentum, wavelets have the potential prescribe an efficient and even way of sampling phase space in this problem.

Another potentially viable approach involved interpreting the discretization of the wavefunction as a representation on grid-point functions, represented with a basis $|g\rangle$ (as opposed to the $|x\rangle$ basis). With this formalism, (55) is replaced with the following:

$$\langle g_f|\psi(t+\Delta t)\rangle = \sum_{p_0}\sum_{g_0} g(g_f - x_{clN})\frac{\exp(iS_{cl}/\hbar)}{h\sqrt{\det(M)}}\tilde{g}(p_0)\langle g_0|\psi(t)\rangle\Delta g\Delta p \qquad (98)$$

where $g(x) \equiv \langle x|g\rangle$ is the grid-point function and $\tilde{g}(p) \equiv \langle p|g\rangle$ is the grid-point function in momentum space. Conceptually, the wavefunction can be thought of as a superposition of these grid-point functions. An individual grid-point function is highly localized in space, implying, by the Heisenberg uncertainty principle, a wide distribution in momenta. This explosion of momenta can be thought to be expressed in the classical paths emanating from the grid-point. These "momenta explosions" is another way of looking at path integrals. The superposition of the resulting explosions of paths gives the new wavefunction.

This grid-point approach was extensively studied early on in the code development (reflected in Appendix A). This approach has the conceptual ease

of being a highly localized virtual classical particle deposit. The best behaved grid-point function found was a Gaussian with a standard deviation of the grid spacing, but all attempts thus far resulted in the high-momentum attenuation discussed in Section B of Chapter IV. Numerous techniques were developed to enhance the momentum distribution to counteract the attenuation, but those techniques were eventually abandoned in this work. However, some applications may find that attenuation acceptable because it has a faster virtual classical particle deposit. This choice is a tradeoff between computation time and accuracy, and the solution presented in Chapter II and III chooses accuracy. An approach using grid-point functions, perhaps in combination with wavelet methods, may yet be found that provides both high accuracy and high efficiency.

The one-particle Hamiltonian used in Chapter II contained an effective potential that was defined in Chapter III to assume a mean-field approximation for the other quantum particles. Higher-order corrections to the effective potential are possible. For the duration of the quantum push, the potential is assumed to be static, which in turn assumes all other quantum particles are static. One could reconstruct the wavefunction, then the fields, at every classical time step, essentially making $\Delta t = \partial t$. A possible correction, without such a drastic increase in CPU time, involves a way to estimate the evolution of the electrostatic potential due to other quantum particles for the duration of the

quantum time step. For the sake of clarity of this presentation, let us assume there is only one other quantum particle. We wish to study the evolution of $|\psi_1\rangle$ of a two particle state $|\psi_1\rangle \quad |\psi_2\rangle$.

$$|\Psi(t)\rangle = \exp(-\frac{i\hat{H}t}{\hbar})|\psi_1\rangle \quad |\psi_2\rangle \tag{99}$$

where the Hamiltonian $\hat{H}$ is of the form

$$\hat{H} = \hat{H}_1 + \hat{H}_{12} + \hat{H}_2 \tag{100}$$

where $\hat{H}_1$ and $\hat{H}_2$ are the part of the Hamiltonian containing operators that act only on particle $|\psi_1\rangle$ and $|\psi_2\rangle$, respectively, and $\hat{H}_{12}$ describes the terms that have operators that act on both particles. Hitting $\langle\psi_2|\exp(\frac{i\hat{H}_2t}{\hbar})$ on (99) gives

$$\langle\psi_2|\exp(\frac{i\hat{H}_2t}{\hbar})|\Psi(t)\rangle = \left(\langle\psi_2|\exp(\frac{i\hat{H}_2t}{\hbar})\exp(-\frac{i\hat{H}_{12}t}{\hbar})\exp(-\frac{i\hat{H}_2t}{\hbar})|\psi_2\rangle\right)\exp(-\frac{i\hat{H}_1t}{\hbar})|\psi_1\rangle \tag{101}$$

Let us assume terms of order $\left(\frac{\hat{H}_{12}t}{\hbar}\right)^2$ and higher are small. Therefore

$$\begin{aligned}&\langle\psi_2|\exp(\frac{i\hat{H}_2t}{\hbar})\exp(-\frac{i\hat{H}_{12}t}{\hbar})\exp(-\frac{i\hat{H}_2t}{\hbar})|\psi_2\rangle\\ &\langle\psi_2|\exp(\frac{i\hat{H}_2t}{\hbar})\left(1-\frac{i\hat{H}_{12}t}{\hbar}\right)\exp(-\frac{i\hat{H}_2t}{\hbar})|\psi_2\rangle\end{aligned} \tag{102}$$

Using

$$\exp(\hat{B})\hat{A}\exp(-\hat{B}) = \hat{A} + \left[\hat{B},\hat{A}\right] + \frac{1}{2}\left[\hat{B},\left[\hat{B},\hat{A}\right]\right] + \frac{1}{3!}\left[\hat{B},\left[\hat{B},\left[\hat{B},\hat{A}\right]\right]\right] + \cdots \tag{103}$$

on (102) yields

$$\begin{aligned}&\langle\psi_2|\exp(\frac{i\hat{H}_2 t}{\hbar})\exp(-\frac{i\hat{H}_{12}t}{\hbar})\exp(-\frac{i\hat{H}_2 t}{\hbar})|\psi_2\rangle \quad 1 - \frac{it}{\hbar}(\\ &\langle\psi_2|\hat{H}_{12}|\psi_2\rangle + \frac{it}{\hbar}\langle\psi_2|\left[\hat{H}_2,\hat{H}_{12}\right]|\psi_2\rangle + \frac{1}{2}\left(\frac{it}{\hbar}\right)^2\langle\psi_2|\left[\hat{H}_2,\left[\hat{H}_2,\hat{H}_{12}\right]\right]|\psi_2\rangle + \cdots)\end{aligned} \tag{104}$$

Let us assume $\hat{H}_i$ and $\hat{H}_{12}$ are of the form

$$\hat{H}_i = \frac{\hat{p}_i^{\,2}}{2m_i} + V_i(\hat{x}_i) \tag{105}$$

$$\hat{H}_{12} = V_{12}(\hat{x}_1,\hat{x}_2) \tag{106}$$

Then, by inserting $\mathbf{1} = \int dx_2|x_2\rangle\langle x_2|$,

$$\langle\psi_2|\hat{H}_{12}|\psi_2\rangle = \int\psi_2^{\,*}(x_2)V_{12}(\hat{x}_1,x_2)\psi_2(x_2)\,dx_2\,, \tag{107}$$

which we recognize as the mean-field approximation. Because

$\left[\hat{x}_1,\hat{x}_2\right] = \left[\hat{x}_1,\hat{p}_2\right] = 0$,

$$\left[\hat{H}_2,\hat{H}_{12}\right] = \frac{1}{2m_2}\left[\hat{p}_2^{\,2},V_{12}(\hat{x}_1,\hat{x}_2)\right] \tag{108}$$

Strategically inserting $\mathbf{1} = \int dx_2|x_2\rangle\langle x_2|$ into $\langle\psi_2|\left[\hat{H}_2,\hat{H}_{12}\right]|\psi_2\rangle$ yields

$$\langle\psi_2\left|\left[\hat{H}_2,\hat{H}_{12}\right]\right|\psi_2\rangle =$$
$$\frac{1}{2m_2}\int\left(\langle\psi_2|\hat{p}_2|x_2\rangle\langle x_2|\hat{p}_2V_{12}(\hat{x}_1,\hat{x}_2)|\psi_2\rangle-\langle\psi_2|V_{12}(\hat{x}_1,\hat{x}_2)\hat{p}_2|x_2\rangle\langle x_2|\hat{p}_2|\psi_2\rangle\right)dx_2 \tag{109}$$

Using $\langle x_2|\hat{p}_2=\frac{\hbar}{i}\frac{\partial}{\partial x_2}\langle x_2|$,

$$\langle\psi_2\left|\left[\hat{H}_2,\hat{H}_{12}\right]\right|\psi_2\rangle =$$
$$\frac{\hbar^2}{2m_2}\int\left(\frac{\partial\psi_2{}^*(x_2)}{\partial x_2}\frac{\partial}{\partial x_2}\left(V_{12}(\hat{x}_1,x_2)\psi_2(x_2)\right)-\frac{\partial}{\partial x_2}\left(\psi_2{}^*(x_2)V_{12}(\hat{x}_1,x_2)\right)\frac{\partial\psi_2(x_2)}{\partial x_2}\right)dx_2 \tag{110}$$

Simplifying the derivatives, canceling terms, and factoring gives

$$\langle\psi_2\left|\left[\hat{H}_2,\hat{H}_{12}\right]\right|\psi_2\rangle=\frac{\hbar^2}{2m_2}\int\frac{\partial V_{12}(\hat{x}_1,x_2)}{\partial x_2}\left(\frac{\partial\psi_2{}^*(x_2)}{\partial x_2}\psi_2(x_2)-\psi_2{}^*(x_2)\frac{\partial\psi_2(x_2)}{\partial x_2}\right)dx_2$$

(111)

We recognize that the derivative terms on ψ_2 is proportional to the probability current and the derivative of V_{12} is a dipole potential. Combining (111), (107), and (104) with (101) gives

$$\langle\psi_2\left|\exp(\frac{i\hat{H}_2t}{\hbar})\right|\Psi(t)\rangle=\exp(-\frac{i\hat{H}_{1_{eff}}t}{\hbar})\left|\psi_1\right\rangle \tag{112}$$

where

$$\hat{H}_{1eff} = \frac{\hat{p}_1^{\,2}}{2m} + V_{1eff}(\hat{x}_1) \tag{113}$$

is the effective Hamiltonian for particle 1 and

$$\begin{aligned} V_{1_{eff}}(\hat{x}_1) &= V_1(\hat{x}_1) + \int \psi_2^*(x_2) V_{12}(\hat{x}_1, x_2) \psi_2(x_2)\, dx_2 \\ &+ \frac{it\hbar}{2m_2} \int \frac{\partial V_{12}(\hat{x}_1, x_2)}{\partial x_2} \left(\frac{\partial \psi_2^*(x_2)}{\partial x_2} \psi_2(x_2) - \psi_2^*(x_2) \frac{\partial \psi_2(x_2)}{\partial x_2} \right) dx_2 + \cdots \end{aligned} \tag{114}$$

is the effective potential for particle 1. In the case of the potentials describing electrostatics, this method evaluates the electric field and potential due to the charge density and dipole components of the other wavefunction. By extending the derivation to higher terms in (104), quadrupole and higher-order components can be included, if desired. Because the terms beyond the original mean-field term are time-dependent terms, this method can be thought of as a predictor-type correction to the mean-field approximation.

An avenue of investigation worth pursuing concerns the momentum Jacobian $\left(\frac{\partial p_{cl\,f}}{\partial p_0} \right)$ that was assumed to be one in producing (55) from (51). While it is most likely that virtual classical particle contributions of neighboring momenta make the cancellation due to phase as smooth as we have seen, it is possible that a correct evaluation of this momentum Jacobian will enable methods that sample momentum space with lesser density possible. In the course of this work, the properties of this multiplier were not explored. It may

also provide ways of estimating the contributions of virtual classical particles of momenta in between virtual classical particles that are evaluated.

In addition, the questions concerning boundary conditions raised in Section D of Chapter II could be addressed. The bounce point of the virtual classical particles being one-half grid beyond the wall is a question that may be answered by a more detailed examination of the semiclassical methods and how they incorporate boundary conditions. For now, the quantum PIC code functions, but, eventually, this issue may be worth investigating.

If the memory limitations could be overcome, this application of the semiclassical method could be used to evaluate systems with large Hilbert spaces (e.g., improvements on (54)), as is the case with wavefunctions of multiple particles. The network of paths traced by the virtual classical particles could be used to link representations of multiparticle quantum wavefunctions, just as they have been here with single particle wavefunctions. In this author's opinion, applying this method of evolving such a wavefunction is viable. The particle pusher would be largely unchanged, while the wavefunction reconstruction routine would evaluate slices (in x_1, x_2, x_3, etc.) of the much larger dataset representing the initial and final multiparticle wavefunctions. If such a code could be constructed, it could simulate the most challenging problems in quantum physics involving quantum correlations between particles. Such a code could model EPR pairs, quantum teleportation, and many

other aspects of quantum computing.

F. Conclusion

In conclusion, we have seen the techniques, theoretical and computational, used to construct, test, and operate a quantum PIC code based on semiclassical methods. The examples shown in this dissertation provide a demonstration of this code's direct applicability to questions in physics where quantum phenomena are important. In addition, this code provides a test bed to further develop and apply semiclassical techniques. Finally, the experience accumulated while developing the code, and the code itself, is ground-breaking work and should be used as a guide in the development of future codes that model quantum mechanics using similar methods, helping to avoid the pitfalls already encountered and suggesting ways of finding reliable and accurate methods. By establishing a foundation for this form of semiclassical methods, this code and its development supply building blocks and knowledge to the development of quantum-mechanical models in the future. This endeavor, like others in science and in art, is "never finished, only abandoned."

VIII. Appendix A

- Development of a New Code

A. Experimentation

Like many other pursuits, discovery and progress in science do not always proceed in a straight line, sometimes encountering problems and dead-ends. But, with patience and time, the correct path can be found, sometimes because of what was learned by encountering such errors. This work is no exception. There was a great deal that went wrong which led us to what was right. Before finding the basic design that is common today, Thomas A. Edison tested hundreds of devices intended to be light bulbs. In this Appendix, we present some of our bad light bulbs, in the hopes that others may learn as we did.

B. Virtual Particle Distribution

An early idea of much discussion was the distribution of the virtual classical paths. In a plasma code, the classical particle data is retained and updated throughout the life of the simulation. Some properties of quantum wavefunctions were known to be like classical plasmas, so the idea was proposed that the classical information in this quantum PIC code would also be retained throughout the life of the simulation. (This idea is expressed in Figure 4 of Reference 11.) It took some time to create the tests that most clearly resolved the consequences of using this approach.

The following comparison uses a pair of electrostatically repulsive quantum particles initialized as Gaussians. These runs were identical except for the nature of the classical particle preparation. One run has the classical information initialized with random positions and momenta, just like in a plasma code. This information at the end of one quantum time step was used for the next. The value of the initial wavefunction that these classical particles acquired was interpolated between grid points like how the plasma code interpolates electric field and potential information. In the other run, the classical information was reinitialized on a space and momentum grid as described in Chapter III. All other aspects of the code were identical. Figure 35 shows the comparison.

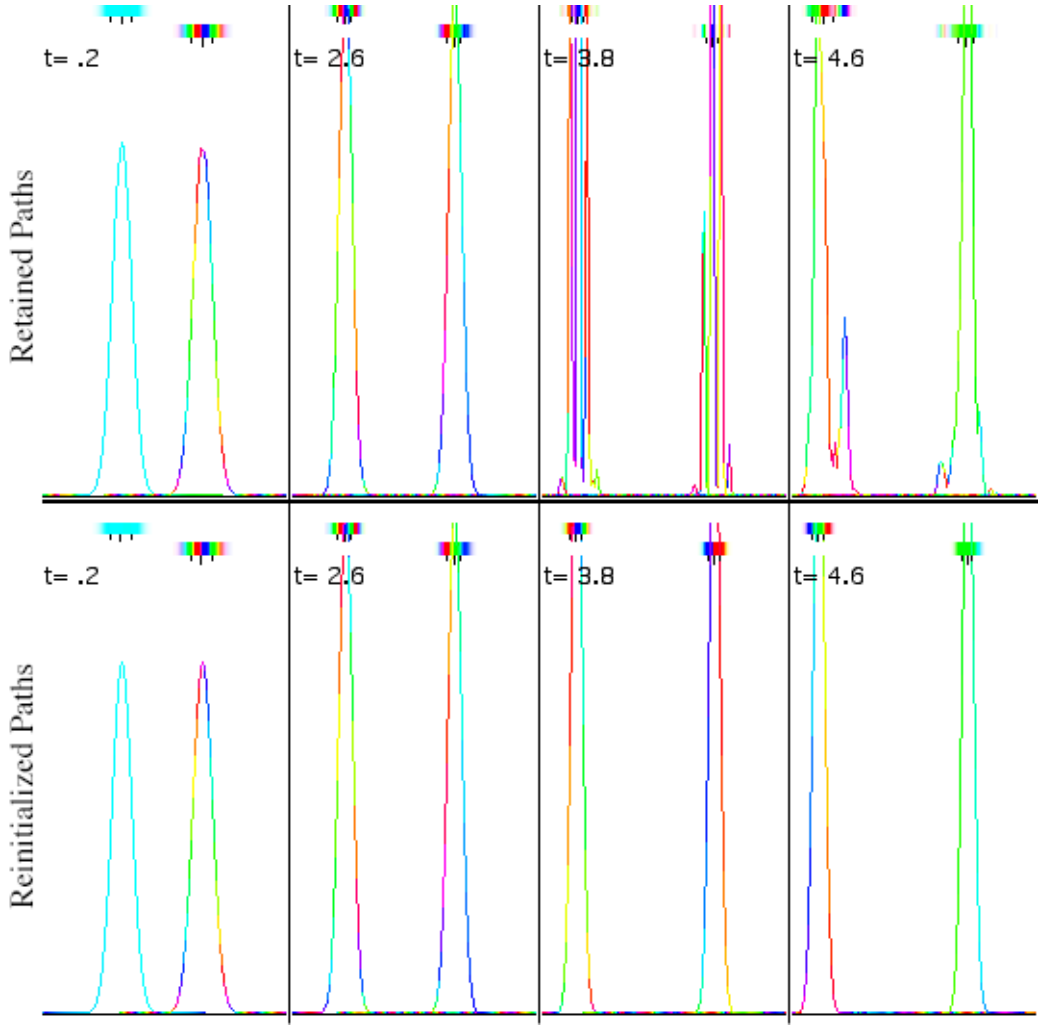


Figure 35. A comparison of runs using randomly initialized classical information retained between time steps (top) versus regularly initialized classical information at every time step (bottom).

Within a couple dozen time steps the result using the plasma-like classical paths shows the wavefunction being shredded (see t=3.8 in the figure). This non-physical behavior is serious enough that the simulation never fully recovers, while the run using classical data that is reinitialized at every time step shows

smooth behavior over many time steps.

This behavior occurs because of a few likely reasons. First, the classical paths, because of how long they are run, could become overpopulated in some parts of phase space, while leaving other portions undersampled. Second, the nature of the random initialization could cause noise from this random arrangement to seep into the quantum wavefunction. This behavior is much like the difference between Monte Carlo integration and other analytical techniques. The stability and correctness of the wavefunction evolution intimately rely on the delicate cancellation of the virtual classical paths' contributions. In some ways, it is a wonder that all of these semiclassical calculations do not degenerate into what is shown in the top of this figure.

C. Alternative Depositing

The initial versions of the quantum PIC code were based on a grid-point representation of the wavefunction. Using grid-point functions were considered a viable approach to representing and understanding a discretized quantum wavefunction. Much of the calculation was unchanged from that shown in Chapter II and III, as a comparison between (98) and (55) will show. Simulations using this method did show qualitative features not unlike theory, and, in particular cases such as the SHO, some quantitative comparisons

showed great promise.

However, as discussed in Section B of Chapter III, an energy loss in the simulation was noticed, studied, heard, and analyzed. The root of the problem was found to be in the virtual classical particle deposit. The net effect of the problem was as if the wavefunction was convoluted with the grid-point function every time step, which was, in retrospect, exactly what the code was doing.

Therefore, we attempted revisions in the code to reverse the undesired effect. In the context of the grid-point representation, the grid-point functions that formed the initial wavefunction were thought to "emit" virtual classical particles, while the grid-point functions forming the final wavefunction were thought to "collect" these particles. Hence, the grid-points at the beginning and end were called "emitters" and "collectors", respectively, a nomenclature inspired by leads of a transistor.

Numerous ideas were attempted, many of which were variations on the functions for the emitter and collector grid-point functions. The emitter functions were implemented in the particle preparation routine, while the wavefunction reconstruction / particle deposit routine handled the collectors. Many of these ideas were not explored chronologically in this order:

- The plasma code's charge deposit - The original algorithm for depositing charge on a grid in the plasma code was still present, and it served the

plasma code well, so we borrowed it for the earliest attempts in the virtual classical particle deposit. It was among the worst high-momentum attenuators.

- Gaussian grid-point functions - Since a Gaussian is the only wavefunction known to possess the minimum uncertainty allowed by the Heisenberg uncertainty principle, it would sample the smallest piece of phase space possible, making this a natural choice. The most obvious choice for the standard deviation σ of the Gaussians was 0.5 grids. This setup demonstrated an energy loss that was easy to predict analytically, which allowed us to confirm that this was the origin of the energy loss. Experiments with a variety of σ from 0.3 to 0.75 were attempted in the hopes of discovering a value that minimized the energy loss. A minimum loss for some test cases were found, but it was not a minimum for all.
- Nearest neighbor - Virtual classical particles contribute only to the grid point closest to its final position. This solution produced stable results, but the energy loss was greater.
- Wide spread of contributions - Since it was conceivable that a virtual classical particle could contribute to more than one grid-point, its contributions were spread, with coefficients determined by the grid-point function, to up to seven grids at a time. This algorithm increased

the computation time without significant other benefit.

- Momentum boosting in the emitters - What if the emitters could compensate for the loss in the collectors? Since the particle distribution from each grid-point ranged evenly in momentum space, it was a simple matter to insert a multiplier as a function of initial momentum, enhancing the momentum distribution. The problem that arose was that it often enhanced the noise as well, and the errors would grow exponentially, destroying the wavefunction. Combinations of other emitters were attempted with other collectors. Also, this approach had the conceptual problem of implying that the emitter and collector functions were not alike, even though that they should represent the same basis. However, it did appear that the effects of the emitters and collectors were independent of each other.
- Post-collector processing - After the virtual classical particles were deposited and the wavefunctions were reconstructed, a tridiagonal solver was performed to “un-convolute” the wavefunction, in the hopes of undoing the “damage” caused to the wavefunction. It helped, but not in a way that was completely consistent with the attenuation.
- Custom-shaped grid-point functions and other functional relationships - Attempts were made to determine grid-point functions whose Fourier transforms were flat for much of momentum space, then tapered off in

the range of momenta that "don't matter". Some of these were based various combinations of exponentials. The idea was: perhaps the attenuation was good to keep for the sake of stability, but a middle range of momenta could be preserved to retain the desired physics. Some of the functions attempted included instances of the Fermi-Dirac distribution function from statistics. Two problems occurred: 1. the resulting function in position space was far more costly to determine and compute to justify keeping; and 2. The "interesting range" of momenta was found to be most of the possible momenta, so the tapering would be far too sharp to do any good.

- Grid-point functions that were zero at neighboring grid-points - Perhaps a grid-point function could be constructed that was zero at integer grid-points away, for example one like $\sin(\pi x)/x$. Using transcendentals proved very costly computationally without significant benefit, and the function $\sin(\pi x)/x$ was not localized enough (i.e., did not decay fast enough) to be practical as a grid-point function. Some were tried based on polynomials that achieved these conditions. Others were designed based on superpositions of three to seven Gaussians, each at positions one grid-point away. The idea was that a pair of Gaussians one grid away from the center would subtract the center one just enough to make the function zero at one grid-point away. Then two more would

make the function zero another grid away, and so on. The coefficients of these Gaussians were adjusted so that these conditions would hold. These tests were combined with a variety of σ 's for the Gaussians, and the emitters were made to match. This produced the best and most stable results, reducing the energy loss from 1 part in 100 to 1 part in 5000 per time step.

For a long time, a loss of 1 part in 5000 for most wavefunctions was the best available. It was only realized later that extending the derivation to its fullest extent led to the equations shown in Chapter II and III. Interpreted in the above context, it described a emitter that was a delta function in space and a collector that was a delta function in momentum. These are perhaps most extreme functions possible by comparison to the above, but it was clear these choices provided an even spread in their corresponding dual spaces.

While the form described by (55) was attempted and found to be successful, the form of (51) seemed suggestive of a deposit in momentum space. Such a deposit was attempted, and the FFT of the plasma code was used to solve for the wavefunction in position space. Some of the same deposit functions used for the position basis collector methods were used for the momentum basis here, but it was clear that such solutions were not enough to adequately describe the momentum ket given by (51). Part of the problem was

that the final momentum of the particle was usually in between grids in momentum space, so the proper representation on a discretized momentum space required a wide deposit of the form $\sin(\pi p)/p$. This wide momentum deposit was constructed and tested. While this depositor met with some success, it was not pursued further because it quickly became at least as costly as depositing a wave throughout all space. It became clear, finally, that a wave deposited throughout all space was the solution that provided the best results.

In the course of this work, some discussion was encountered concerning the physical justification of a particle affecting all space. A wave arising from a path is analogous to how a ray of light hitting a wall represents an entire wavefront impinging on the surface. At a moment in time, the surface receives light at a distribution of phase depending on the angle of incidence. In this case, the final wavefunction receives the virtual classical particle in an identical distribution of phase depending on its momentum. The paths could be thought to represent the flow of entire wavefronts, rather than simply particles.

Conceptual worries about local effects being maintained despite a completely nonlocal deposit were pacified in light of how close to perfection this solution was. Later, the optimization technique described in Section C of Chapter III was developed, and the wavefunction reconstruction routine has been largely unchanged ever since.

IX. Appendix B

- Quantum PIC Source Code

A. Source Code

To provide a concrete understanding of the structure of the quantum PIC code, its source code is provided here. These listings, in Fortran, contain a large amount of dead code that is switched on and off via comments or if tests, allowing the same code to be used for a variety of experiments.

B. Main Program Loop

```
      program babycq1
      use plib
      implicit none
! indx = exponent which determines length in x direction,
nx=2**indx
```

```
! npx = number of background particles distributed in x
direction
! npxb = number of beam particles per species in x direction
! nspecies = number of species (e.g., quantum particles)
! for monte carlo quantum: np/dx > 250
! for grided quantum: np/dx > 2*h (=128)
      integer :: indx, npx, npxb, nspecies
!      parameter( indx =   7, npxb = 0, nspecies = 1)
!      parameter( indx =   7, npxb = 0, nspecies = 2)
!      parameter( indx =   8, npxb = 0, nspecies = 16)
!      parameter( indx =   8, npxb = 0, nspecies = 16)
      parameter( indx =   11, npxb = 0, nspecies = 64)
!      parameter( indx =   7, npx =   20480, npxb = 0,
nspecies = 1)
!      parameter( indx =   7, npx =   20480, npxb = 0,
nspecies = 2)
!      parameter( indx =   8, npx =   81920, npxb = 0,
nspecies = 1)
!      parameter( indx =   8, npx =   81920, npxb = 0,
nspecies = 16)
!     parameter( indx =  11, npx =  409600, npxb = 0,
nspecies = 1)
!     parameter( indx =  14, npx = 4096000, npxb = 409600,
nspecies =1)
!     parameter( indx=18, npx = 40960000, npxb = 4096000,
nspecies = 1)
! tend = time at end of simulation, in units of plasma
frequency
! dt = time interval between successive classical
calculations
      real :: tend, dt, tcptq, vscale, sigma, div2sigsq
      parameter( tend =  104.000, dt = 0.20000e+00)
      parameter( tcptq = 32, vscale = tcptq)
!      parameter( tcptq = 32, vscale = 32.0)
       parameter( sigma = 0.5, div2sigsq =
0.5/(sigma*sigma))
! vtx = thermal velocity of electrons in x direction
! vdx = drift velocity of beam electrons x direction
! vtdx = thermal velocity of beam electrons in x direction
      real :: vtx, vdx, vtdx, avdx
      parameter( vtx =   1.000, vdx =  0.000, vtdx =  1.00)
! avdx = absolute value of drift velocity of beam electrons
x direction
      parameter( avdx =   5.000)
! npx is determined by maintaining a particle density and a
spread in phase space
      parameter( npx = (2**indx)*(2**indx) )
!      parameter( npx = 10*(2**indx)*(vtx*dt*tcptq*2) )
! indnvp = exponent determining number of real or virtual
```

```
processors
! indnvp must be < indx
! idps = number of partition boundaries
! idimp = dimension of phase space = 2
! mshare = (0,1) = (no,yes) architecture is shared memory
      integer :: indnvp, idps, idimp, mshare, np, nx, nxh,
nloop
      parameter( indnvp =   4, idps =    2, idimp =   8,
mshare =   0)
! np = total number of electrons in simulation
      parameter(np=npx+npxb)
      parameter(nx=2**indx,nxh=nx/2)
! nloop = number of time steps in simulation
      parameter(nloop=tend*vscale/(tcptq*dt)+.0001)
! nvp = number of real or virtual processors, nvp =
2**indnvp
! nblok = number of particle partitions
      integer :: nvp, nblok
      parameter(nvp=2**indnvp,nblok=1+mshare*(nvp-1))
! npmax = maximum number of particles in each partition
! nxpmx = maximum size of particle partition, including
guard cells.
      integer :: npmax, nxpmx
      parameter(npmax=(np/nvp)*1.21+250,nxpmx=(nx-1)/nvp+4)
! kxp = number of complex grids in each field partition
! kblok = number of field paritions
      integer :: kxp, kblok, nbmax, ntmax
      parameter(kxp=(nxh-1)/nvp+1,kblok=1+mshare*(nxh/kxp-
1))
      integer :: kxpc, kblokc
      parameter(kxpc=(nx-1)/nvp+1,kblokc=1+mshare*(nx/kxpc-
1))
! nbmax = size of buffer for passing particles between
processors

parameter(nbmax=1+(2*(npx*vtx+npxb*vtdx)+1.4*npxb*avdx)*dt/n
x)
! ntmax = size of hole array for particles leaving
processors
      parameter(ntmax=2*nbmax)
      complex wfcn, wf, wfl, wfl2, ffc, fc, qc, pc, sct
       complex qkinH, qptot, wkinH, wfp
! wfcn(j,l,k) = wavefunction at j of species l in partition
k
      dimension wfcn(nxpmx,nspecies,nblok)
      real :: pi, twopi, planck, planckbar, divhbar
      parameter(pi=3.14159625358979323846264338327950288)
      parameter(twopi=6.28318530717959)
! planck = Planck's constant
```

```
      parameter(planck=2*tcptq)
!      parameter(planck=64)

parameter(planckbar=planck/(2*pi),divhbar=2*pi/planck)
      complex :: wmult
      real :: part, q, fx, pt
      common /large/ part
! part(1,n,l,k) = position x of particle n of species l in
partition k
! part(2,n,l,k) = velocity vx of particle n of species l in
partition k
! part(3,n,l,k) = action S of particle n of species l in
partition k
! part(4,n,l,k) = magnitude of particle n's piece of species
l in partition k
! part(5,n,l,k) = phase of particle n's piece of species l
in partition k
! part(6,n,l,k) = det of step i of particle n of species l
in partition k
! part(7,n,l,k) = det of step i-1 of particle n of species l
in partition k
      dimension part(idimp,npmax,nblok)
!      dimension part(idimp,npmax,nspecies,nblok)
!      integer :: btree
!      dimension btree(4, npmax, nspecies, nblok)
! q(j,l,k) = charge density l at grid point jj, where jj = j
+ noff(k) - 1
! fx(j,l,k) = force/charge l at grid point jj, that is
convolution of
! electric field over particle shape, where jj = j + noff(k)
- 1
      dimension q(nxpmx,nspecies,nblok),
fx(nxpmx,nspecies,nblok)
      dimension pt(nxpmx,nspecies,nblok)
! qc(j,k) = complex charge density for fourier mode jj - 1
! fc(j,k) = complex force/charge for fourier mode jj - 1
! where jj = j + kxp*(k - 1)
      dimension qc(kxp,kblok), fc(kxp,kblok)
      dimension pc(kxp,kblok)
! ffc = complex form factor array for poisson solver
      dimension ffc(kxp,kblok)
! mixup = array of bit reversed addresses for fft
! sct = sine/cosine table for fft
      integer :: mixup
      dimension mixup(kxp,kblok), sct(kxp,kblok)
      integer, dimension(kxpc,kblokc) :: mixupc
      complex, dimension(kxpc,kblokc) :: sctc
! edges(1,k) = left boundary of particle partition k
! edges(2,k) = right boundary of particle partition k
```

```
      real :: edges
      dimension edges(idps,nblok)
! nxp(k) = number of primary gridpoints in particle
partition k.
! noff(k) = leftmost global gridpoint in particle partition
k.
      integer :: nxp, noff, npp, nps
      dimension nxp(nblok), noff(nblok)
! noffglob(k) = leftmost global gridpoint in processor k.
      integer, dimension(nvp) :: noffglob
! nxpglob(k) = number of primary gridpoints in processor k.
      integer, dimension(nvp) :: nxpglob
! npp(l,k) = number of particles of species l in partition k
! nps(l,k) = starting address of particles of species l in
partition k
      dimension npp(nspecies, nblok), nps(nspecies, nblok)
! sbufl = buffer for particles being sent to left processor
! sbufr = buffer for particles being sent to right processor
      real :: sbufl, sbufr, rbufl, rbufr
      dimension sbufl(idimp,nbmax,nblok),
sbufr(idimp,nbmax,nblok)
! rbufl = buffer for particles being received from left
processor
! rbufr = buffer for particles being received from right
processor
      dimension rbufl(idimp,nbmax,nblok),
rbufr(idimp,nbmax,nblok)
! ihole = location of holes left in particle arrays
      integer :: ihole, jsl, jsr, jss
      dimension ihole(ntmax,nblok)
! jsl(idps,k) = number of particles going left in particle
partition k
! jsr(idps,k) = number of particles going right in particle
partition k
      dimension jsl(idps,nblok), jsr(idps,nblok)
! jss(idps,k) = scratch array for particle partition k
! scr(idps,k) = scratch array for particle partition k
      real :: scr
      dimension jss(idps,nblok), scr(idps,nblok)
  991 format (5h t = ,i7)
  992 format (19h * * * q.e.d. * * *)
  993 format (34h field, kinetic, total energies = ,3e14.7)
! qme = charge on electron, in units of e
! ax = half-width of particle in x direction
!      data qme,ax /-1.,.8666667/
      real :: qme, ax
      data qme,ax /-1.00,.8666667/   !
!      data qme,ax /-0.12500000,.8666667/   !
!      data qme,ax /-0.09973557,.8666667/   ! sqrt(1/8π)/2
```

```
!      data qme,ax /-0.1994711402,.8666667/   ! sqrt(1/8π)
!      data qme,ax /-0.25,.8666667/
!      data qme,ax /-0.5,.8666667/
!      data qme,ax /-1.0,.8666667/
      integer :: nproc, lgrp, mreal, mint, mcplx
      common /pparms/ nproc, lgrp, mreal, mint, mcplx

      integer, parameter :: cxexpsize = 1024   ! table for
wdeposit
      complex, dimension(cxexpsize+1+cxexpsize+1) :: cxexpt

      integer, parameter :: deltarestartindex = 10   !
number of quantum pushes between rewrites


! Variables used in main
      integer :: kstrt, k, l, itime, idproc, j, joff, isign,
nxp3, lt
      integer :: kw, kt, nextrestartindex, ierr, msid, lstat
      parameter (lstat = 8)
      integer, dimension(lstat) :: istatus
      real, dimension(nspecies) :: initialPosition,
initialMomentum
      real :: anx, qtme, affp, zero, qi0, etime, we, pkx,
wke, wt
      real :: bcoeff, ccoeff, dcoeff, avex, avesqx, wptH,
qptH, qiw
      real :: avedet, avedet2
      complex :: tempCx

! initialize for parallel processing
!      write (6,*) 'initializing PP'
      print *, 'initializing PP'
      call ppinit(idproc,nvp)
      open(unit=6,file='output1-
'//char(48+(idproc/10))//char(48+idproc-10*(idproc/10)), &
     &
FORM="FORMATTED",STATUS="UNKNOWN",POSITION="APPEND")
       kstrt = idproc + 1
! initialize timer
      call timera(-1,'total   ',etime)
! initialize constants
      itime = 0
      anx = float(nx)
      qtme = qme*dt
      affp = anx/float(np*nspecies)
      zero = 0.

      cxexpt(1) = -1
```

```
!      sigma = sqrt(0.15) = 0.387298335
!      sigH = sigma+0.02
!      sigL = sigma-0.02
!      wLastH = 0.

       bcoeff = -exp(-div2sigsq)/(1+exp(-4*div2sigsq))
      ccoeff = 0.0
      bcoeff = 0.0

      if (.false.) then
       bcoeff = exp(-div2sigsq)*((1+exp(-2*div2sigsq))**2)*&
      &(exp(-2*div2sigsq)-1-exp(-4*div2sigsq))/&
      &(1+exp(-4*div2sigsq)+2*exp(-6*div2sigsq)+exp(-
8*div2sigsq)+exp(-12*div2sigsq))
       ccoeff = exp(-2*div2sigsq)/&
      &(1+exp(-4*div2sigsq)+2*exp(-6*div2sigsq)+exp(-
8*div2sigsq)+exp(-12*div2sigsq))
       end if
      dcoeff = 0.0

      if (.false.) then
       bcoeff = -exp(-div2sigsq)*(1+exp(-2*div2sigsq)+exp(-
4*div2sigsq)+exp(-6*div2sigsq)&
      &+2*exp(-8*div2sigsq)+3*exp(-10*div2sigsq)+2*exp(-
12*div2sigsq))/(&
      &(1+exp(-4*div2sigsq)+2*exp(-6*div2sigsq)+2*exp(-
8*div2sigsq)+&
      &2*exp(-10*div2sigsq)+exp(-12*div2sigsq)+exp(-
16*div2sigsq))&
      &*(1+exp(-8*div2sigsq)) )
       ccoeff = exp(-2*div2sigsq)*(1+exp(-2*div2sigsq)+exp(-
4*div2sigsq))/&
      &(1+exp(-4*div2sigsq)+2*exp(-6*div2sigsq)+2*exp(-
8*div2sigsq)+&
      &2*exp(-10*div2sigsq)+exp(-12*div2sigsq)+exp(-
16*div2sigsq))
       dcoeff = -exp(-3*div2sigsq)/&
      &(1+exp(-4*div2sigsq)+2*exp(-6*div2sigsq)+3*exp(-
8*div2sigsq)+&
      &2*exp(-10*div2sigsq)+2*exp(-12*div2sigsq)+3*exp(-
16*div2sigsq))
       end if

! calculate partition variables
      call dcomp1(edges,nxp,noff,nx,kstrt,nvp,idps,nblok)
! for distributed mpi - assuming fixed partition edges
      call
MPI_ALLGATHER(noff,1,mint,noffglob,1,mint,lgrp,ierr)
```

```
      call
MPI_ALLGATHER(nxp,1,mint,nxpglob,1,mint,lgrp,ierr)
!      write (6,*) 'noff(:)',noffglob, 'nxp(:)',nxpglob

      if (.true.) then
       noffglob(1+idproc) = noff(1)
       nxpglob(1+idproc) = nxp(1)
        do kw=0,nvp-1       ! This loop is structured this
way so we don't get a
         if (kw.gt.0) then  ! compiler error: underflowed
register count
          kt = 1 + idproc - kw
          if (kt.lt.1) kt = kt + nvp
   ! recieve into the right spot
          call MPI_IRECV(noffglob(kt),1,mint,kt-
1,kw,lgrp,msid,ierr)
          kt = 1 + idproc + kw
          if (kt.gt.nvp) kt = kt - nvp
   ! send data to the one who needs it
          call MPI_SEND(noff(1),1,mint,kt-1,kw,lgrp,ierr)
          call MPI_WAIT(msid,istatus,ierr)

          kt = 1 + idproc - kw
          if (kt.lt.1) kt = kt + nvp
   ! recieve into the right spot
          call MPI_IRECV(nxpglob(kt),1,mint,kt-
1,kw,lgrp,msid,ierr)
          kt = 1 + idproc + kw
          if (kt.gt.nvp) kt = kt - nvp
   ! send data to the one who needs it
          call MPI_SEND(nxp(1),1,mint,kt-1,kw,lgrp,ierr)
          call MPI_WAIT(msid,istatus,ierr)

         end if
        end do
!      write (6,*) 'noff(:)',noffglob, 'nxp(:)',nxpglob
      end if


! prepare fft tables
      isign = 0
      call
pfft1r(qc,fc,isign,mixup,sct,indx,kstrt,kxp,kblok)
      call
pfft1c(qc,fc,isign,mixupc,sctc,indx,kstrt,kxpc,kblokc)
! calculate form factors
      call ppois1
(qc,fc,isign,ffc,ax,affp,we,nx,kstrt,kxp,kblok)
! initial density profile and maxwellian velocity
```

```
distribution
! background electrons
      write (6,*) 'init background es'
      do 120 k = 1, nblok
        do l=1,nspecies
          nps(l,k) = 1
          npp(l,k) = 0
        enddo
      nxp3 = nxp(k) + 3
  120 continue
!      if (npx.gt.0) call pistr1
(part,edges,npp,nps,vtx,zero,npx,nx,    &
!     &idimp,npmax,nblok,idps,nspecies)
! beam electrons
      do 140 k = 1, nblok
        do l=1,nspecies
          nps(l,k) = npp(l,k) + 1
        enddo
  140 continue
!      if (npxb.gt.0) call pistr1
(part,edges,npp,nps,vtdx,vdx,npxb,nx,  &
!     &idimp,npmax,nblok,idps,nspecies)

! Initialize wavefunctions as normalized Gaussians moving at
various velocities
      write (6,*) 'Initializing wavefunctions'

      if (.true.) then
        call
classicalICs(initialPosition,initialMomentum,nspecies,nx,  &
     &    planck)
      end if

      qi0 = sqrt(0.01/pi)
      do k=1,nblok
!        nxp3 = nxp(k) + 3
        joff = noff(k) - 2
        do l=1,nspecies
          pkx = twopi*initialMomentum(l)
!          pkx = 2*pi*(8.5-l)*2/(nspecies*6)
!          pkx = 2*pi*0/nx
!          pkx = -0.06125*(l-1.5)
!          pkx = 0.5 - 1.1*2.0*(l-0.5*(1+nspecies))/nspecies
!          pkx = 0
!       print *, 'first zero ',8-joff
          do j=1,nxpmx
            if (.false.) then
              wfcn(j,l,k) = exp(-0.25/(4**2)*(j + joff -
(l+0.5)*nx/(nspecies+2) + 0)**2) *&
```

```
     &qi0 * cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff)))
            end if

            if (.true.) then
              wfcn(j,l,k) = exp(-0.25/(4**2)*(j + joff -
initialPosition(l))**2) *&
     &qi0 * cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff)))
            end if

            if (.false.) then
              wfcn(j,l,k) = exp(-0.01*(j + joff -
((l)*nx/(nspecies+1) + 0) )**2) *&
     &qi0 * cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff)))
            end if

            if (.false.) then
              wfcn(j,l,k) = sin( (j + joff -
2.0)*(1.0)*twopi*0.5/(nx-4) )*&
     &exp(-0.0001*(j + joff - ((l)*nx/(nspecies+1) + 0)
)**2) *&
     &qi0 * cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff)))
            end if

            if (.false.) then
              wfcn(j,l,k) = sin( (j + joff -
2.0)*(l+6)*twopi*0.5/(nx-4) ) * qi0
            end if

            if (.false.) then
              tempCx = 0
              do lt=1,5
                tempCx = tempCx + sin( (j + joff -
2.0)*(lt)*twopi*0.5/(nx-4) ) *&
     & (1.0/lt) * cmplx(cos(twopi*(0.2*lt)*(2*l-3)),
sin(twopi*(0.2*lt)*(2*l-3)))
              end do
              wfcn(j,l,k) = tempCx * qi0
            end if

             if (.false.) then
              wfcn(j,l,k) = sin( (j + joff -
2.0)*(1.0)*twopi*0.5/(nx-4) )*&
     &exp(-0.01*(j + joff - ((l+3)*nx/(nspecies+6) + 0)
)**2) *&
     &qi0 * cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff)))
            end if

             if (.false.) then
              wfcn(j,l,k) = exp(-0.01*(j + joff -
```

```
((l+3)*nx/(nspecies+5)) )**2) *&
     &qi0 * cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff)))
            end if

             if (.false.) then
              if (j.eq.1) print *,
(twopi/(256.0*(2.0**(1.0/3))))
              wfcn(j,l,k) = exp(-
(twopi/(256.0*(2.0**(1.0/3))))*(j + joff -
((l+1)*nx/(nspecies+3)) )**2) *&
     &qi0 * cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff)))
            end if

             if (.false.) then
              if (j.eq.1) print *,
(twopi/(256.0*(2.0**(1.0/3))))
              wfcn(j,l,k) = exp(-
(twopi/(256.0*(2.0**(1.0/3))))*(j + joff -
((l+1)*nx/(nspecies+3)) )**2) *&
     &qi0 * cmplx(cos(pkx*(j+joff)), sin(pkx*(j+joff))) * (j
+ joff - (nx/2))
            end if

           if (.false.) then
             wfcn(j,l,k) = cmplx(cos(pkx*(j+joff)),
sin(pkx*(j+joff))) / nx
            end if

            if (.false.) then
              if ( (j + joff).eq.((l+2)*nx/(nspecies+1+4)) )
then
                   wfcn(j,l,k) = 1.0
              else
                wfcn(j,l,k) = 0
              end if
            end if

            if (.false.) then
              qi0 = (j + joff - 1 - nx/2)
              pkx = qi0*qi0*divhbar*(1.0/8.0)
              wfcn(j,l,k) = exp(-
(1.0/8.0)*0.5*divhbar*qi0**2) * &
     &(1 + qi0 *(.25 - 0.02*((4*pkx-20)*pkx+15) +
0.004*(((8*pkx-84)*pkx+210)*pkx-105) ) )
!     &(1 + 0*qi0 *(0.75 - 0.2*((4*pkx-20)*pkx+15) +
0.01*(((8*pkx-84)*pkx+210)*pkx-105) ) )
            end if

!            write (6,*) wfcn(j,l,k)
```

```
          enddo
        enddo
      enddo

! calculate background ion density
!     qi0 = -qme/affp
!      wmult = sqrt( divhbar * vscale / (2 * pi * dt *
tcptq)) * nx / np &
!     &* cmplx(cos(0.25*pi),sin(-0.25*pi))
!      wmult = tcptq * vtx * dt * nx / real(np)  &
!     &* cmplx(cos(0*pi),sin(-0*pi))
      wmult = nx / ( real(np) ) &
     &* cmplx(cos(0*pi),sin(-0*pi))
      write (6,*) 'npx= ', npx, '  h= ', planck
      write (6,*) wmult,divhbar, bcoeff, ccoeff, dcoeff
      write (6,*) 'noff(:)',noffglob, 'nxp(:)',nxpglob


      do k = -5, 4
        qiw = 0.5*k + 0.2
        j = qiw + .5
        print *, qiw, j
      end do

      write (6,*) 'opening wavefunction file'

      nextrestartindex = deltarestartindex
      if (prepareifrestarting(nloop,nspecies,nx,idproc,nvp,&
     &     wfcn,noffglob,nxpglob,nblok,nxpmx,itime)) then

        nextrestartindex = nspecies * itime +
deltarestartindex/2
      else
      if (idproc.eq.0) then

!Open wavefunction file
      open (unit=9,file="outputW",status="REPLACE")
      write (9,*) 'NSTEPS=', nloop
      write (9,*) 'NQ=',nspecies
      write (9,*) 'NX=',nx
      write (9,*) 'START'

      call openbinarywf(nloop,nspecies,nx,idproc,nvp)

      end if
      end if

!
! * * * start main iteration loop * * *
```

```
!
      write (6,*) 'begin data'
  500 if (nloop.le.itime) go to 2000
      if (kstrt.eq.1) write (6,991) itime
      print *,'tstep',itime,'/',nloop
      wke = 0.
       qkinH = 0.
       qptH = 0.
       qptot = 0.

!      sigma = 0.5*(sigH+sigL)
!      write (6,*) 'sigma = ', sigma
!      div2sigsq = 0.5/(sigma*sigma)

      do k = 1, nblok
        do j = 1, nxpmx
          do l=1,nspecies
! initialize charge density to zero
            q(j,l,k) = 0.0
          enddo
        enddo
      enddo
! deposit charge   using qme |wfcn|^2 leaving out self-
energy
      do 1190 k = 1, nblok
! only where we need to
      do 1180 j = 1, nxp(k)
      do l=1,nspecies
      qiw = qme*(real(wfcn(j+1,l,k))**2 +
aimag(wfcn(j+1,l,k))**2)
        do lt=1,nspecies
          if (l.ne.lt) then
            q(j+1,lt,k) = q(j+1,lt,k) + qiw
          end if
        enddo
      enddo
 1180 continue
 1190 continue

      do l=1,nspecies
      print *,'Computing Fields'

! transform charge to fourier space
      isign = -1
! copy data from particle to field partition, and add up
guard cells
      call cppfp1
(q,qc,isign,scr,kstrt,nvp,nxpmx,nblok,kxp,kblok,idps, &
     &l,nspecies)
```

```
      call
pfft1r(qc,fc,isign,mixup,sct,indx,kstrt,kxp,kblok)
! calculate force/charge in fourier space
      call ppois1
(qc,fc,isign,ffc,ax,affp,we,nx,kstrt,kxp,kblok)
! transform force/charge to real space
      isign = 1
      call ppois1
(qc,pc,isign,ffc,ax,affp,we,nx,kstrt,kxp,kblok)
      call
pfft1r(fc,qc,isign,mixup,sct,indx,kstrt,kxp,kblok)
      call
pfft1r(pc,qc,isign,mixup,sct,indx,kstrt,kxp,kblok)
! copy data from field to particle partition, and copy to
guard cells
      call cppfp1
(fx,fc,isign,scr,kstrt,nvp,nxpmx,nblok,kxp,kblok,idps,&
     &l,nspecies)
      call cppfp1
(pt,pc,isign,scr,kstrt,nvp,nxpmx,nblok,kxp,kblok,idps,&
     &l,nspecies)
! particle push and charge density update
!     call timera(-1,'push    ')

      if (.true.) call
addexternalpot(fx,pt,l,noff,itime,qtme,dt,vscale,tcptq,nxpmx
,nblok,nspecies,nx)

      if (.true.) then
      print *,'Preparing particles'
! initialize particles
      call
pprepw(wfcn,part,npp,noff,nxp,vtx,vscale,divhbar,dt,   &

&l,sigma,nx,npx,idimp,npmax,nblok,nxpmx,nspecies,idproc,nvp)

      print *,'Pushing particles'
! push particles
!        if (.false.) then
!      print *,'preparing btree'
!      call
preparebt(part,btree,npp,noff,idimp,npmax,nblok,nxpmx, &
!     &l,nspecies)
!        end if
      do k=1,tcptq
!        if (.false.) then
!      print *,'  push'
!      call
ppush1bt(part,fx,npp,noff,qtme,dt,wke,idimp,npmax,nblok,nxpm
```

```
x, &
!     &nx,l,nspecies,pt,btree)
!        else
      call
ppush1(part,fx,npp,noff,qtme,dt,wke,idimp,npmax,nblok,nxpmx,
&
     &nx,l,nspecies,pt)
!        end if
! move particles into appropriate spatial regions
      call pmove1
(part,edges,npp,sbufr,sbufl,rbufr,rbufl,ihole,jsr,jsl,&

&jss,nx,kstrt,nvp,idimp,npmax,nblok,idps,nbmax,ntmax,ierr,l,
&
     &nspecies)
      enddo

      print *,'Depositing wavefunction'
! push wavefunctions
      call
wdeposit(wfcn,part,npp,noff,nxp,indx,l,divhbar,wmult,dt,
&

&div2sigsq,idimp,npmax,nblok,nvp,idproc,nxpmx,nspecies,vscal
e,  &
     &noffglob,nxpglob,kxpc,kblokc,mixupc,sctc,cxexpt)

      else

      qi0 = sqrt(0.02/pi)
      do k=1,nblok
        joff = noff(k) - 1
          pkx = 1.0*(l-1.0)
          do j=1,nxpmx
            wfcn(j,l,k) = exp(-
(0.01/cmplx(1,(1+itime)*dt*planckbar*0.02))*&
     &(j + joff - l*nx/(nspecies+1))**2) *qi0
          enddo
      enddo

      end if

      print *,'Running diagnostics'
! Run diagnostics and renormalize
!      qi0 = qme
      call
wdiagn(wfcn,noff,nxp,qiw,avex,avesqx,wkinH,wptH,wfp,pt,&

&planckbar,div2sigsq,vscale,l,npmax,nblok,nxpmx,nspecies,idp
```

```
roc,nvp,&
     &idimp,part,npp,avedet,avedet2,nx,qme)

      write (6,*) l,' size is ',qiw,' at ',avex,' width
',sqrt(avesqx-avex*avex)
      write (6,*) l,' avedet is ',avedet,'  width
',sqrt(avedet2-avedet*avedet)
      write (6,*) ' kin, pot, H, p
',wkinH,wptH,wkinH+wptH,wfp

      if (.false.) then
      write (9,*) itime, l
      do k = 1, nblok
        do j = 1, nxp(k)
          write (9,*) wfcn(j+1,l,k)
        end do
        write (8) wfcn(:,l,k)
      end do

       endfile 9
       backspace 9
       else

       call
wcollatendump(wfcn,noffglob,nxpglob,nblok,nxpmx,itime,l,nspe
cies,idproc,nvp)

       end if

      qkinH = qkinH + wkinH
      qptH = qptH + wptH
      qptot = qptot + wfp

       do k=1,nblok
       write (6,*) k, npp(l,k)
       enddo

       endfile 6
       backspace 6
! call flush(9)

       if (.not.(real(wfcn(2,1,1)).gt.real(wfcn(3,1,1))))
then
       if (.not.(real(wfcn(2,1,1)).lt.real(wfcn(3,1,1))))
then
       if (.not.(real(wfcn(2,1,1)).eq.real(wfcn(3,1,1))))
then
! we might have a NaN
         write (6,*) 'NaN?'
```

```
           itime = nloop
           call MPI_ABORT(lgrp, 255, ierr)
           exit
         endif
         endif
         endif

       end do

       if (ierr.eq.1) then
       write (6,*) 'init background es'
       do k = 1, nblok
         do l=1,nspecies
           nps(l,k) = 1
           npp(l,k) = 0
         enddo
       nxp3 = nxp(k) + 3
       enddo
       if (npx.gt.0) call pistr1
(part,edges,npp,nps,vtx,zero,npx,nx,    &
     &idimp,npmax,nblok,idps,nspecies)
       endif

!      call timera(1,'push    ',etime)
! energy diagnostic, now meaningless
       wt = we + wke
       if (kstrt.eq.1) then
          write (6,993) we, wke, wt
           write (6,*) 'qkin, qpt, Htot, ptot', qkinH, qptH,
qkinH+qptH, qptot
       endif

        if ((itime*nspecies).ge.nextrestartindex) then
          nextrestartindex = itime*nspecies +
deltarestartindex

         if (idproc.eq.0) then
! flush does not exist, it seems
           endfile 9
           backspace 9

!           endfile 8
!           backspace 8
           call fuflush()

           write (6,*) 'Writing restart info file'
           endfile 6
           backspace 6
```

```
          open (unit=11, file='restartinfo',
status="REPLACE")

          write (11,*) itime
          write (11,*) nextrestartindex

          endfile 11

          close(unit=11)
        end if

         print *, '  Press Command-. to stop simulation'

       end if

      itime = itime + 1
      go to 500
 2000 continue
!
! * * * end main iteration loop * * *
!
      if (idproc.eq.0) then
        write (9,*) 'END'

        call fuclose()

        if (itime.ge.nloop) then ! it's the end, protect the
data
          open (unit=11, file='restartinfo',
status="REPLACE")

          write (11,*) itime
          write (11,*) -1

          close(unit=11)
        end if

      end if

      if (kstrt.eq.1) write (6,992)
      call timera(1,'total   ',etime)
      call ppexit
!      pause
      stop
      end program
```

Listing A. A listing of the main program of the quantum PIC code.

```
      subroutine
addexternalpot(fx,pt,l,noff,itime,qtm,dt,vscale,tcptq,nxpmx,
nblok,nspecies,nx)
! for 1d code, this subroutine adds an external potential to
the pt and fx arrays
! in space, with periodic boundary conditions, for
distributed data.
! fx(j,k) = force/charge at grid point jj, that is
convolution of
! electric field over particle shape, where jj = j + noff(k)
- 1
! noff(k) = leftmost global gridpoint in particle partition
k.
! qtm = particle charge/mass ratio times dt
! dt = time interval between successive calculations
! nblok = number of particle partitions.
! nxpmx = maximum size of particle partition, including
guard cells.
! scalar version with spatial decomposition
      implicit none
      integer :: l, itime, nblok, nxpmx, nspecies, nx
      real :: qtm, dt, vscale, tcptq
      real, dimension(nxpmx,nspecies,nblok) :: fx, pt
      integer, dimension(nblok) :: nxp, noff
      real :: omegasq, slope, adjustment, rtemp, height
      integer :: j, k, joff, xt, width
      parameter(omegasq = (1.0/8.0)**2, slope = 1.0/4.0,
width = 4, height=16.0)

! This accounts for the unusual units that pt and fx take
      adjustment = dt/(qtm*(vscale**2))

      do k=1,nblok
        joff = noff(k) - 2
        do j=1,nxpmx
          if (.false.) then
!            ! Simple Harmonic Oscillator
            xt = j + joff - nx/2
            pt(j,l,k) = pt(j,l,k) +
adjustment*0.5*omegasq*(xt**2)
            fx(j,l,k) = fx(j,l,k) - adjustment*omegasq*xt
          end if

          if (.false.) then
!            ! time dependent ramp
```

```
            if ((itime*dt).ge.1) then
              xt = j + joff - nx/2
              rtemp = -1*exp(-0.25*(itime*dt-1))
              pt(j,l,k) = pt(j,l,k) + adjustment*rtemp*xt
              fx(j,l,k) = fx(j,l,k) - adjustment*rtemp
            end if
          end if

          if (.false.) then
!           ! Triagular barrier or well
            xt = (j + joff - nx/2)
            if (abs(xt).le.width) then
              if (xt.lt.0) then
                pt(j,l,k) = pt(j,l,k) +
adjustment*slope*(xt+width)
                fx(j,l,k) = fx(j,l,k) - adjustment*slope
              else
                pt(j,l,k) = pt(j,l,k) +
adjustment*slope*(width-xt)
                fx(j,l,k) = fx(j,l,k) + adjustment*slope
              end if
            end if
          end if

          if (.false.) then
!           ! Rectangular barrier or well
            xt = (j + joff - nx/2)
            if (abs(xt).le.width) then
             if (abs(xt).gt.(width-2)) then
              if (xt.lt.0) then
                pt(j,l,k) = pt(j,l,k) +
adjustment*height*0.5*(xt+width)
                fx(j,l,k) = fx(j,l,k) -
adjustment*height*0.5
              else
                pt(j,l,k) = pt(j,l,k) +
adjustment*height*(width-xt)
                fx(j,l,k) = fx(j,l,k) +
adjustment*height*0.5
              end if
             else
                pt(j,l,k) = pt(j,l,k) + adjustment*height
             end if
            end if
          end if

          if (.false.) then
!           Side walls    ?
            xt = (j + joff - nx/2)
```

```
            if (abs(xt).ge.(nx/2 - 3)) then
              if (xt.lt.0) then
                pt(j,l,k) = pt(j,l,k) -
adjustment*64*0.5*((nx/2 - 3) + xt)
                fx(j,l,k) = fx(j,l,k) + adjustment*64*0.5
              else
                pt(j,l,k) = pt(j,l,k) -
adjustment*64*0.5*((nx/2 - 3) - xt)
                fx(j,l,k) = fx(j,l,k) - adjustment*64*0.5
              end if
            end if
          end if

          if (.false.) then
!            ! Simple Harmonic Oscillator potentials
            xt = j + joff - l*nx/(nspecies+1)
            pt(j,l,k) = pt(j,l,k) +
adjustment*0.5*omegasq*(xt**2)
            fx(j,l,k) = fx(j,l,k) - adjustment*omegasq*xt
          end if

          if (.false.) then
!            ! 1-D atom potentials
            xt = j + joff - nx/2
              if (xt.lt.0) then
                pt(j,l,k) = pt(j,l,k) -
adjustment*slope*(xt)
                fx(j,l,k) = fx(j,l,k) + adjustment*slope
              else if (xt.gt.0) then
                pt(j,l,k) = pt(j,l,k) +
adjustment*slope*(xt)
                fx(j,l,k) = fx(j,l,k) - adjustment*slope
              end if
          end if


        enddo
      enddo

      end subroutine
```

Listing B. The external potential routine.

D. Particle Preparation

```
      subroutine
pprepw(wfcn,part,npp,noff,nxp,vtx,vscale,divhbar,dt,   &

&l,sigma,nx,npx,idimp,npmax,nblok,nxpmx,nspecies,idproc,nvp)
! for 1d code, this subroutine stores wavefunction density
! using second-order spline interpolation, with periodic
boundaries
! and distributed data into the particle array.
! density is approximated by values at the nearest grid
points
! q(n)=qm*(.75-dx**2), q(n+1)=.5*qm*(.5+dx)**2, q(n-
1)=.5*qm*(.5-dx)**2
! where n = nearest grid point and dx = x-n
! part(1,n,l,k) = position x of particle n of species l in
partition k
! q(j,l,k) = species l density at grid point jj, where jj =
j + noff(k)-1
! wfcn = given complex valued wavefunction
! part = particle data
! npp(l,k) = number of particles of species l in partition k
! noff(k) = leftmost global gridpoint in particle partition
k.
! idimp = size of phase space + action + old position
! npmax = maximum number of particles in each partition
! nblok = number of particle partitions.
! nvp = number of (virtual) processors.
! nxpmx = maximum size of particle partition, including
guard cells.
! complex scalar version with spatial decomposition
      implicit none
! common block for parallel processing
      integer nproc, lgrp, lstat, mreal, mint, mcplx
! lstat = length of status array
      parameter(lstat=8)
! lgrp = current communicator
! mreal = default datatype for reals
      common /pparms/ nproc, lgrp, mreal, mint, mcplx

      integer :: npp, noff, l, idimp, npmax, nblok, nxpmx,
nspecies
      integer :: nx, npx, idproc, nvp
      real :: part, vtx, vscale, divhbar, dt, sigma, bcoeff,
ccoeff, dcoeff
      complex, dimension(nxpmx,nspecies,nblok) :: wfcn
      dimension part(idimp,npmax,nblok)
!      dimension part(idimp,npmax,nspecies,nblok)
      dimension npp(nspecies,nblok), noff(nblok)
      integer, dimension(nblok) :: nxp
      complex :: wf, wfr, wfl
```

```
      real :: dx, siginterp, rtemp, pi
      integer :: k, j, ip, nn, nnoff, kr, kl, msid, istatus,
ierr
      dimension istatus(lstat)
      parameter(pi = 3.14159265358979323846264338327950288)

       siginterp = 0.25/(sigma*sigma)

      if (.true.) then  ! zero edges

       if (.false.) then
        ! Broadcast left & right edges
        if (idproc.eq.0) then
          wfl = wfcn(4,l,1)
        end if
        if (idproc.eq.(nvp-1)) then
          wfr = wfcn(-1+nxp(nblok),l,nblok)
        end if

        call MPI_BCAST(wfl,1,mcplx,0,lgrp,ierr)
        call MPI_BCAST(wfr,1,mcplx,nvp-1,lgrp,ierr)

        ! subtract off to make it zero at edges
        wf = (wfl-wfr)/(nx-5)

        write (6,*) 'Left, right, slope', wfl, wfr, wf

        do k = 1, nblok
          nnoff = noff(k) - 4
          do j = 1, nxpmx
            wfcn(j,l,k) = wfcn(j,l,k) + (j + nnoff)*wf - wfl
          end do
        end do

       end if

       if (.false.) then
        if (idproc.eq.0) then
          wfcn(1,l,1) = - wfcn(17,l,1)
          wfcn(2,l,1) = - wfcn(16,l,1)
          wfcn(3,l,1) = - wfcn(15,l,1)
          wfcn(4,l,1) = - wfcn(14,l,1)
          wfcn(5,l,1) = - wfcn(13,l,1)
          wfcn(6,l,1) = - wfcn(12,l,1)
          wfcn(7,l,1) = - wfcn(11,l,1)
          wfcn(8,l,1) = - wfcn(10,l,1)
          wfcn(9,l,1) = cmplx(0.,0.)
!          wfcn(2,l,1) = 3 * wfcn(5,l,1) - 8 * wfcn(4,l,1) !
cmplx(0.,0.)
```

```
!          wfcn(3,l,1) = wfcn(5,l,1) - 3 * wfcn(4,l,1)    !
make first & second derivatives continuous
!          wfcn(4,l,1) = cmplx(0.,0.)
         end if
         if (idproc.eq.(nvp-1)) then
!          wfcn(-1+nxp(nblok),l,nblok) = cmplx(0.,0.)
!          wfcn(0+nxp(nblok),l,nblok) = wfcn(nxp(nblok)-
2,l,nblok) - 3 * wfcn(nxp(nblok)-1,l,nblok)   ! make first &
second derivatives continuous
!          wfcn(1+nxp(nblok),l,nblok) = 3 * wfcn(nxp(nblok)-
2,l,nblok) - 8 * wfcn(nxp(nblok)-1,l,nblok)   !cmplx(0.,0.)
           wfcn(-7+nxp(nblok),l,nblok) = cmplx(0.,0.)
           wfcn(-6+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
8,l,nblok)
           wfcn(-5+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
9,l,nblok)
           wfcn(-4+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
10,l,nblok)
           wfcn(-3+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
11,l,nblok)
           wfcn(-2+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
12,l,nblok)
           wfcn(-1+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
13,l,nblok)
           wfcn(0+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
14,l,nblok)
           wfcn(1+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
15,l,nblok)
           wfcn(2+nxp(nblok),l,nblok) = - wfcn(nxp(nblok)-
16,l,nblok)
           wfcn(3+nxp(nblok),l,nblok) = cmplx(0.,0.)
         end if
       end if

         if (idproc.eq.0) then
           wfcn(1,l,1) = cmplx(0.,0.)
           wfcn(2,l,1) = cmplx(0.,0.)
           wfcn(3,l,1) = cmplx(0.,0.)
           wfcn(4,l,1) = cmplx(0.,0.)
          if (.true.) then
           wfcn(5,l,1) = cmplx(0.,0.)
           wfcn(6,l,1) = cmplx(0.,0.)
          if (.false.) then
           wfcn(7,l,1) = cmplx(0.,0.)
           wfcn(8,l,1) = cmplx(0.,0.)
           wfcn(9,l,1) = cmplx(0.,0.)
           wfcn(10,l,1) = cmplx(0.,0.)
!           wfcn(11,l,1) = cmplx(0.,0.)
          end if
```

```
         end if
        end if
        if (idproc.eq.(nvp-1)) then
         if (.true.) then
         if (.false.) then
!          wfcn(-7+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-6+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-5+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-4+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-3+nxp(nblok),l,nblok) = cmplx(0.,0.)
         end if
          wfcn(-2+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-1+nxp(nblok),l,nblok) = cmplx(0.,0.)
         end if
          wfcn(0+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(1+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(2+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(3+nxp(nblok),l,nblok) = cmplx(0.,0.)
        end if
      end if

      do k=1,nblok
!  copy wfcn to guard cells

        kl = k + idproc - 1
        if (kl.lt.1) then
           kl = kl + nvp
        end if

        kr = k + idproc + 1
        if (kr.gt.nvp) then
          kr = kr - nvp
        end if

! for shared
!        wfcn(nxp(kl)+2,l,kl) = wfcn(2,l,k)
!        wfcn(nxp(kl)+3,l,kl) = wfcn(3,l,k)
!        wfcn(1,l,kr) = wfcn(nxp(k)+1,l,k)
! for mpi distributed
        call MPI_IRECV(wfcn(1,l,k),1,mcplx,kl-
1,0,lgrp,msid,ierr)
        call MPI_SEND(wfcn(nxp(k)+1,l,k),1,mcplx,kr-
1,0,lgrp,ierr)
        call MPI_WAIT(msid,istatus,ierr)

        call MPI_IRECV(wfcn(nxp(k)+2,l,k),2,mcplx,kr-
1,1,lgrp,msid,ierr)
        call MPI_SEND(wfcn(2,l,k),2,mcplx,kl-1,1,lgrp,ierr)
        call MPI_WAIT(msid,istatus,ierr)
```

```
      end do

! Calculate initial wavefunction by particle
      do k = 1, nblok

! Reinitialize to grid of trajectories; reuse variables
        npp(l,k) = npx/nvp           ! Num particles per
block
        nn = nx/nvp - 1              ! Grids per block - 1
        nnoff = npx/nx                 ! Particles per space
density
        do j = 0, nn                   ! Loop over position
          do ip = 1, nnoff             !  Loop over initial
momentum
            part(1,j*nnoff+ip,k) = j + noff(k)
            part(2,j*nnoff+ip,k) = vtx*(
&
    &          2.0*real(ip-0.5)/real(nnoff) - 1.0)
!!            part(2,j*nnoff+ip,k) = vtx*ranorm(0)
          end do
        end do

       nnoff = noff(k) - 2
       do j = 1, npp(l,k)
! find interpolation weights for initial position
        nn = part(1,j,k) + .5
        dx = part(1,j,k) - float(nn)
        nn = nn - nnoff
! Reset action
        part(3,j,k) = 0.0
!     wf =((.75 - dx*dx)*wfcn(nn,l,k) +
.5*(wfcn(nn+1,l,k)*(.5 + dx)**2+&
!    &wfcn(nn-1,l,k)*(.5 - dx)**2)) *
cmplx(cos(phase),sin(phase))*wmult
! interpolate and save the wavefunction's ...
        wfl = wfcn(nn-1,l,k)
        wf = wfcn(nn,l,k)
        wfr = wfcn(nn+1,l,k)

! magnitude and ...
!      part(4,j,k) = ((.75 - dx*dx)*abs(wf)
+.5*(abs(wfr)*(.5 + dx)**2 &
!     &     + abs(wfl)*(.5 - dx)**2))
      if (.true.) then      ! dx.eq.0
        if (.true.) then

          if (.false.) then
        part(4,j,k) = abs(wf)              &
```

```
     &   * exp(-0.5*(
((((sigma*vscale*part(2,j,k)*divhbar)**2)**2)  &
     &                  **2) ) )
          end if

          if (.true.) then
           if (.false.) then
! booster
             rtemp = abs(vscale*part(2,j,k)*divhbar)
             if (rtemp.gt.(1.0)) rtemp = 1.0
             wf = wf * ( rtemp * rtemp / ( 12 *
((sin(rtemp/6))**2) *    &
     &                (1 + 2 * cos(rtemp/3)) ) )
           end if
! high p taper
           if (.true.) then
        rtemp = 2.5*(abs(vscale*part(2,j,k)*divhbar) - 2.5 )
        part(4,j,k) = abs(wf)                &
     &   / (1.0 + exp( 4*rtemp ) )
           end if
          end if

          if (.false.) then
        part(4,j,k) = abs(wf)                &
     &   * exp(-0.5*((sigma*vscale*part(2,j,k)*divhbar)**2))
&
      &   * (1 + 2*(bcoeff *cos(vscale*part(2,j,k)*divhbar)
+ &
      &   ccoeff *cos(2*vscale*part(2,j,k)*divhbar) + &
      &   dcoeff *cos(3*vscale*part(2,j,k)*divhbar) ) )
!     &   * (1 - (exp(-0.5/(sigma*sigma))/(1 + exp(-
2.0/(sigma*sigma)))  &
!     &   * 2*cos(vscale*part(2,j,k)*divhbar)) )
          end if

          if (.false.) then
            rtemp = abs(vscale*part(2,j,k)*divhbar)
            if (rtemp.le.pi) then
              part(4,j,k) = abs(wf)
            else
              part(4,j,k) = abs(wf)*exp(6*(pi**2 -
rtemp**2))
            end if
          end if

! phase separately
           part(5,j,k) = atan2(aimag(wf),real(wf))
         else
      part(4,j,k) = (exp(-siginterp*dx*dx)*abs(wf)    &
```

```
      &    + abs(wfr)*exp(-siginterp*(dx - 1)**2)    &
     &    + abs(wfl)*exp(-siginterp*(1 + dx)**2) )
&
     &   * exp(-0.5*((sigma*vscale*part(2,j,k)*divhbar)**2))
        if (wf.eq.0) then
          wf = wfr + wfl
        end if
! phase separately
        wfr = wfr * conjg(wf)
        wfl = wfl * conjg(wf)
!        part(5,j,k) = atan2(aimag(wf),real(wf)) + &
!     &    (.5*(atan2(aimag(wfr),real(wfr))*(.5 + dx)**2 + &
!     &     atan2(aimag(wfl),real(wfl))*(.5 - dx)**2))
        part(5,j,k) = atan2(aimag(wf),real(wf)) + &
     &     atan2(aimag(wfr),real(wfr))*exp(-siginterp*(dx -
1)**2) + &
     &     atan2(aimag(wfl),real(wfl))*exp(-siginterp*(1 +
dx)**2)
         end if
!        wf = q * cmplx(cos(phase),sin(phase))
      end if

           part(6,j,k) = 1.0   ! det of step 0
           part(7,j,k) = 0.0   ! det of "step -1"
           part(8,j,k) = 0.0   ! unused (for now)

       end do
      end do

      end subroutine
```

Listing C. Particle preparation routine.

E. Particle Push

```
      subroutine
ppush1(part,fx,npp,noff,qtm,dt,ek,idimp,npmax,nblok,   &
     &nxpmx,nx,l,nspecies,pt)
! for 1d code, this subroutine updates particle co-ordinate
and velocity
! using leap-frog scheme in time and second-order spline
interpolation
! in space, with periodic boundary conditions, for
distributed data.
```

```
! equations used are:
! v(t+dt/2) = v(t-dt/2) + (q/m)*fx(x(t))*dt, where q/m is
charge/mass,
! and x(t+dt) = x(t) + v(t+dt/2)*dt
! fx(x(t)) is approximated by interpolation from the nearest
grid points
! fx(x) = (.75-dx**2)*fx(n)+.5*(fx(n+1)*(.5+dx)**2+fx(n-
1)*(.5-dx)**2)
! where n = nearest grid point and dx = x-n
! part(1,n,l,k) = position x of particle n of species l in
partition k
! part(2,n,l,k) = velocity vx of particle n of species l in
partition k
! fx(j,k) = force/charge at grid point jj, that is
convolution of
! electric field over particle shape, where jj = j + noff(k)
- 1
! npp(l,k) = number of particles of species l in partition k
! noff(k) = leftmost global gridpoint in particle partition
k.
! qtm = particle charge/mass ratio times dt
! dt = time interval between successive calculations
! kinetic energy/mass at time t is also calculated, using
! ek = .125*sum((v(t+dt/2)+v(t-dt/2))**2)
! idimp = size of phase space = 2
! npmax = maximum number of particles in each partition
! nblok = number of particle partitions.
! nxpmx = maximum size of particle partition, including
guard cells.
! nx = size of space.
! scalar version with spatial decomposition
      implicit none
      integer ::
idimp,npmax,nspecies,nblok,nxpmx,npp,noff,l,nx
      real :: sum1, work1, part, fx, pt, qtm, dt, ek
      dimension part(idimp,npmax,nblok)
!      dimension part(idimp,npmax,nspecies,nblok)
      dimension
fx(nxpmx,nspecies,nblok),npp(nspecies,nblok),noff(nblok)
      dimension pt(nxpmx,nspecies,nblok)
      dimension sum1(1), work1(1)

      integer :: j,k, nn, nnoff
      real :: dx, ax, px, phase, oldposition
      real, parameter :: bounceposition = 1.5

      sum1(1) = ek*8.
      do 20 k = 1, nblok
      nnoff = noff(k) - 2
```

```
      do 10 j = 1, npp(l,k)

      phase = 0.0 ! Initialize phase adjustment
      oldposition = part(1,j,k) ! store just in case
! find interpolation weights
      nn = part(1,j,k) + .5
      dx = part(1,j,k) - float(nn)
      nn = nn - nnoff
! find acceleration
      ax = (.75 - dx*dx)*fx(nn,l,k) + .5*(fx(nn+1,l,k)*(.5 +
dx)**2 +   &
     &fx(nn-1,l,k)*(.5 - dx)**2)
      px = (.75 - dx*dx)*pt(nn,l,k) + .5*(pt(nn+1,l,k)*(.5 +
dx)**2 +   &
     &pt(nn-1,l,k)*(.5 - dx)**2)
! new velocity
      dx = part(2,j,k) + qtm*ax
! average kinetic energy
      sum1(1) = sum1(1) + (part(2,j,k) + dx)**2
! action accumulate
!      part(3,j,k) = part(3,j,k) + .125*dt*(part(2,j,k) +
dx)**2 -
!     &qtm*px
      part(3,j,k) = part(3,j,k) + .5*dt*((dx)**2) - qtm*px
! new velocity
      part(2,j,k) = dx
! new position
      part(1,j,k) = part(1,j,k) + dx*dt

      if (.true.) then
! check if beyond boundary & reflect
      if (part(1,j,k).lt.(bounceposition)) then
!        part(1,j,k) = (bounceposition)*2 - part(1,j,k)
        part(1,j,k) = oldposition
        if (part(2,j,k).lt.(0.0)) part(2,j,k) = -
part(2,j,k)
!        phase = phase +
3.14159265358979323846264338327950288
      else if (part(1,j,k).gt.(nx-bounceposition)) then
!        part(1,j,k) = (nx-bounceposition)*2 - part(1,j,k)
        part(1,j,k) = oldposition
        if (part(2,j,k).gt.(0.0)) part(2,j,k) = -
part(2,j,k)
!        phase = phase +
3.14159265358979323846264338327950288
      end if
      end if

! find interpolation weights
```

```
      nn = part(1,j,k) + .5
      dx = part(1,j,k) - float(nn)
      nn = nn - nnoff
! push det's, using (∂t)^2 V'' / m = (dt/vscale)^2
(qtm*(vscale**2)/dt)*pt'' / m = pt''*qtm*dt
      px = pt(nn+1,l,k) + pt(nn-1,l,k) - 2.0*pt(nn,l,k)
      ax = part(7,j,k)
      part(7,j,k) = part(6,j,k)
      part(6,j,k) = (2.0 - px*qtm*dt) * part(7,j,k) - ax
! now to check for possible behavior of the det
      px = part(7,j,k)*part(6,j,k)
      if (px.lt.0) then    ! we have a zero crossing (sign
flip)
        phase = phase -
3.14159265358979323846264338327950280*0.5
      else if (px.eq.0) then
        phase = phase -
3.14159265358979323846264338327950280*0.25
      end if

! update phase with any changes
      part(5,j,k) = part(5,j,k) + phase

   10 continue
   20 continue

! this line is used for distributed memory mpi computers
      call psum(sum1,work1,1,1)
! normalize kinetic energy
      ek = .125*sum1(1)
      end subroutine
```

Listing D. Particle push routine.

F. Wavefunction Reconstruction / Particle Deposit

```
      subroutine
wdeposit(wfcn,part,npp,noff,nxp,indx,l,divhbar,wmult,dt,  &

&div2sigsq,idimp,npmax,nblok,nvp,idproc,nxpmx,nspecies,vscal
e,      &
     &noffglob,nxpglob,kxp,kblok,mixup,sct,cxexpt)
! for 1d code, this subroutine distributes wavefunction
density
```

```
! using second-order spline interpolation, with periodic
boundaries
! and distributed data.
! density is approximated by values at the nearest grid
points
! q(n)=qm*(.75-dx**2), q(n+1)=.5*qm*(.5+dx)**2, q(n-
1)=.5*qm*(.5-dx)**2
! where n = nearest grid point and dx = x-n
! part(1,n,l,k) = position x of particle n of species l in
partition k
! q(j,l,k) = species l density at grid point jj, where jj =
j + noff(k)-1
! wfcn = given complex valued wavefunction; output returned
here
! part = particle data
! npp(l,k) = number of particles of species l in partition k
! noff(k) = leftmost global gridpoint in particle partition
k.
! noffglob(k) = leftmost global gridpoint in particle
partition k (all partitions).
! divhbar = inverse of hbar
! idimp = size of phase space + action + old position
! npmax = maximum number of particles in each partition
! nblok = number of particle partitions.
! nxpmx = maximum size of particle partition, including
guard cells.
! complex scalar version with spatial decomposition
      implicit none
! common block for parallel processing
      integer nproc, lgrp, lstat, mreal, mint, mcplx
! lstat = length of status array
      parameter(lstat=8)
! lgrp = current communicator
! mreal = default datatype for reals
      common /pparms/ nproc, lgrp, mreal, mint, mcplx
! get definition of MPI constants
      include 'mpif.h'

! Information needed for using the fft
      integer :: kxp, kblok, isign, kstrt, kfinish
      integer, dimension(kxp,kblok) :: mixup
      complex, dimension(kxp,kblok) :: sct
      integer :: kdep, kdeprange
      parameter (kdeprange = 64)

      integer :: npp, noff, l, idimp, npmax, nblok, nxpmx
      integer :: indx, nx, nspecies, nvp, idproc
      real :: part, divhbar, vscale, div2sigsq
      complex, dimension(nxpmx,nspecies,nblok) :: wfcn
```

```
      dimension part(idimp,npmax,nblok)
!      dimension part(idimp,npmax,nspecies,nblok)
      dimension npp(nspecies,nblok), noff(nblok)
      integer, dimension(nvp) :: noffglob
      integer, dimension(nvp) :: nxpglob
      integer, dimension(nblok) :: nxp
! I need the full space, even in dist. mem
      complex, dimension(nxpmx, nvp, nblok) :: wtemp
      complex :: wmult, wf, ctemp, cincr
      real :: dx, dt, phase, pi, pdh, wavenn, rtemp
      integer :: k, j, kw, jw, nn, nnoff, kt, msid, istatus,
ierr, itemp
      dimension istatus(lstat)
      parameter ( pi = 3.14159265358979323846264338327950288
)

      complex, dimension(kxp, nvp, kblok) :: wktemp
      complex, dimension(kxp, kblok) :: f, g
      integer, dimension(nvp) :: blocklengths
      logical, parameter :: doPdeposit = .false.

      logical, parameter :: usecxexpt = .false.
      integer, parameter :: cxexpsize = 1024
      complex, dimension(cxexpsize+1+cxexpsize+1) :: cxexpt

      nx = 2**indx

!      print *, indx, nx, kxp, kblok

       if (.not.doPdeposit) then ! write wave
!Set wtemp to zero
      do k=1,nblok
       do kw=1,nvp
        do j=1,nxpmx
          wtemp(j,kw,k) = 0.0
        end do
       end do
      end do

        if (usecxexpt) then
         if (real(cxexpt(1)).lt.0) then
           ! needs initialization
           do j=0,cxexpsize+1
             cxexpt(j) = cmplx(cos((j-
1)*pi/cxexpsize),sin((j-1)*pi/cxexpsize))
             cxexpt(j+cxexpsize+1) = cmplx(cos((j-
1)*2*pi/(cxexpsize*cxexpsize)),sin((j-
1)*2*pi/(cxexpsize*cxexpsize)))
           end do
```

```
         end if
        end if

       else ! using p deposit
!Set wktemp to zero
      do k=1,kblok
       do kw=1,nvp
        do j=1,kxp
          wktemp(j,kw,k) = 0.0
        end do
       end do
      end do

       end if


! Deposit by particle path
      do k = 1, nblok
       nnoff = noff(k) - 2
      do j = 1, npp(l,k)

! p final divided by hbar
        pdh = vscale*part(2,j,k)*divhbar
!        if (.false.) then
!        if (part(2,j,l,k).lt.0.0) then
!          pdh = vscale*(2.0 - part(2,j,l,k))*divhbar
!        end if
!        else
        if (doPdeposit) then ! for p deposit (or equiv)
! find interpolation weights for final momentum, & adjust
for negative values
          dx =  pdh * (nx/(2*pi)) + 2*nx
          nn = dx + .5
          dx =  dx - float(nn)
          nn = nn - 2*nx
!          pdh = nn * (2*pi/nx)
        end if

!       Total phase =  Action/hbar   + atan2(original psi)
        phase = vscale * part(3,j,k) * divhbar + part(5,j,k)
! Vaslov factors et al to be accounted for elsewhere

! Additional phase:   - pfinal * xend / hbar
        phase = phase - pdh * part(1,j,k)

! Read initial wf out of particle memory & multiply
!   mechanism for getting contribution to final psi
!        wf = wmult * part(4,j,l,k) *
cmplx(cos(phase),sin(phase))
```

```
!        wf = wmult * part(4,j,l,k)     ! w/o determinant
factor
        if (abs(part(6,j,k)).gt.0.00001) then
          wf = wmult * part(4,j,k) / sqrt(abs(part(6,j,k)-
part(7,j,k)))  ! w/ det factor
        end if

! Now we distribute into the next wavefunction

! Information from one particle is distributed to enitre
wavefunction, but with phase shift
       if (.not.doPdeposit) then
        do kw=1,nvp
         nnoff = noffglob(kw) - 2
!         nnoff = ((kw-1)*nx)/nvp - 2
         if (.true.) then

             ctemp = wf * &
     &  cmplx(cos(phase + pdh * ( nnoff + 1 ) ), &
     &        sin(phase + pdh * ( nnoff + 1 ) ) )
           cincr = cmplx( cos(pdh), sin(pdh) )
           do jw=1,nxpmx     !  2,nxpglob(kw)+1

             wtemp(jw,kw,k) = wtemp(jw,kw,k) + ctemp

             ctemp = ctemp * cincr
           end do

         else
         do jw=1,nxpmx     !  2,nxpglob(kw)+1
           itemp = nnoff + jw
!           if (itemp.ge.112) itemp = itemp - nx

           if (usecxexpt) then
             rtemp = (phase + pdh * ( itemp ) ) * (0.5/pi)
             itemp = rtemp + 0.5
             if ( (rtemp+0.5).lt.0) itemp = itemp - 1
             rtemp = rtemp - float(itemp)

             ! -0.5 <= rtemp < 0.5

             itemp = (2*cxexpsize)*abs(rtemp)
             ctemp = cxexpt(itemp+1)
             if (rtemp.lt.0) then
               ctemp = wf * conjg(ctemp)
             else
               ctemp = wf * ctemp
             end if
```

```
            else
            ctemp = wf * &
     &  cmplx(cos(phase + pdh * ( itemp ) ), &
     &        sin(phase + pdh * ( itemp ) ) )
            end if

            wtemp(jw,kw,k) = wtemp(jw,kw,k) + ctemp
          end do

          end if
         end do

        else ! Assuming p space

!         wavenn = (2*pi/nx) * nn
         if ((nn.lt.(nx/2)).and.(nn.ge.(-nx/2))) then
           if (abs(dx).lt.(0.00001)) then
             ctemp = 1.0
           else
             ctemp = sin(pi*dx)/(pi*dx)
           end if
           ctemp = ctemp * wf * &
     &       cmplx(cos(phase + pi*dx),sin(phase + pi*dx))

           jw = nn
           if (nn.lt.0) jw = jw + nx
           kw = jw/kxp
           jw = 1 + jw - kxp*kw
!          if ((jw.gt.0).and.(jw.le.kxp)) then
           wktemp(jw, 1+kw, k) = wktemp(jw, 1+kw, k) + ctemp
!          else
!           print *, jw, kw
!          end if
         end if

         !equiv of
!         do kw=1,nvp
!          nnoff = noffglob(kw) - 2
!          do jw=2,nxpglob(kw)+1
!            wf = ctemp * &
!     &  cmplx(cos(wavenn * ( nnoff + jw ) ), &
!     &        sin(wavenn * ( nnoff + jw ) ) )
!            wtemp(jw,kw,k) = wtemp(jw,kw,k) + wf
!          end do
!         end do

! higher p's
!         wavenn = (2*pi/nx)*(nn+1)
```

```
        kstrt = 1
        kfinish = kdeprange
        if ((nn+kstrt).lt.(-nx/2)) kstrt = - (nx/2) - nn
        if ((nn+kfinish).ge.(nx/2)) kfinish = (nx/2) - nn -
1
        do kdep = kstrt, kfinish
          jw = nn + kdep
!          if ((jw.lt.(nx/2)).and.(jw.ge.(-nx/2))) then
            ctemp = sin(pi*(dx-kdep))/(pi*(dx-kdep)) * wf  &
     & * cmplx(cos(phase + pi*(dx-kdep)),sin(phase + pi*(dx-
kdep)))

            if (jw.lt.0) jw = jw + nx
            kw = jw/kxp
            jw = 1 + jw - kxp*kw
!           if ((jw.gt.0).and.(jw.le.kxp)) then
            wktemp(jw, 1+kw, k) = wktemp(jw, 1+kw, k) +
ctemp
!           else
!            print *, jw, kw
!           end if
!          end if
        end do

! lower p's
!        wavenn = (2*pi/nx)*(nn-1)
        kstrt = 1
        kfinish = kdeprange
        if ((nn-kstrt).ge.(nx/2)) kstrt = 1 - (nx/2) + nn
        if ((nn-kfinish).lt.(-nx/2)) kfinish = (nx/2) + nn
        do kdep = kstrt, kfinish
          jw = nn - kdep
!          if ((jw.lt.(nx/2)).and.(jw.ge.(-nx/2))) then
            ctemp = wf * sin(pi*(dx+kdep))/(pi*(dx+kdep)) *
&
     & cmplx(cos(phase + pi*(dx+kdep)),sin(phase +
pi*(dx+kdep)))

            if (jw.lt.0) jw = jw + nx
            kw = jw/kxp
            jw = 1 + jw - kxp*kw
!           if ((jw.gt.0).and.(jw.le.kxp)) then
            wktemp(jw, 1+kw, k) = wktemp(jw, 1+kw, k) +
ctemp
!           else
!            print *, jw, kw
!           end if
!          end if
        end do
```

```
       end if

      end do
      end do

!copy wtemp pieces to wfcn (note: no guard cells are
necessary)
       if (.false.) then
!  for shared memory
      do k=1,nblok

        do j=1,nxpmx
          wfcn(j,l,k) = wtemp(j,k,k)
        end do

      end do

      do k=1,nblok

        do kw=1,nvp
          if (kw.ne.k) then
           do j=1,nxpmx
             wfcn(j,l,kw) = wfcn(j,l,kw) + wtemp(j,kw,k)
           end do
          end if
        end do

      end do
       end if

       if (.not.doPdeposit) then
!  for distributed mpi
         if (.false.) then
! (Same as MPI_REDUCE_SCATTER, but that's not in MacMPI as
of 980722)
      do k=1,nblok
        ! copy to self
        do j=1,nxpmx
          wfcn(j,l,k) = wtemp(j,k+idproc,k)
        end do

      end do
      ! Now there's room to play with in wtemp

      do k=1,nblok

        do kw=1,nvp-1
          kt = k + idproc - kw
```

```
          if (kt.lt.1) kt = kt + nvp
   ! recieve into the freed up block
          call MPI_IRECV(wtemp(1,k+idproc,k),nxpmx,mcplx,kt-
1,kw,lgrp,msid,ierr)
          kt = k + idproc + kw
          if (kt.gt.nvp) kt = kt - nvp
   ! send data to the one who needs it
          call MPI_SEND(wtemp(1,kt,k),nxpmx,mcplx,kt-
1,kw,lgrp,ierr)
          call MPI_WAIT(msid,istatus,ierr)

           do j=1,nxpmx
             wfcn(j,l,k) = wfcn(j,l,k) + wtemp(j,k+idproc,k)
           end do

        end do

      end do
         else

           do kw = 1, nvp
             blocklengths(kw) = nxpmx
           end do
           call MPI_REDUCE_SCATTER(wtemp(:,:,1),
wfcn(:,l,1), &
     &             blocklengths, mcplx, MPI_SUM, lgrp, ierr)

         end if
       end if

       if (doPdeposit) then ! p deposit
!  for distributed mpi

       if (.false.) then
! (Same as MPI_REDUCE_SCATTER, but that's not in MacMPI as
of 980729)

!      print *, 'summing into f'
      do k=1,kblok
        ! copy to self
        do j=1,kxp
          f(j,k) = wktemp(j,k+idproc,k)
        end do

      end do
      ! There's room to play with in wktemp

      do k=1,kblok
```

```
        do kw=1,nvp-1
          kt = k + idproc - kw
          if (kt.lt.1) kt = kt + nvp
   ! recieve into the freed up block   wktemp(1,k+idproc,k)
          call MPI_IRECV(g(1,k),kxp,mcplx,kt-
1,kw,lgrp,msid,ierr)
          kt = k + idproc + kw
          if (kt.gt.nvp) kt = kt - nvp
   ! send data to the one who needs it
          call MPI_SEND(wktemp(1,kt,k),kxp,mcplx,kt-
1,kw,lgrp,ierr)
          call MPI_WAIT(msid,istatus,ierr)

           do j = 1, kxp
             f(j,k) = f(j,k) + g(j,k)
!             wktemp(j,k+idproc,k) = wktemp(j,k+idproc,k) +
wk2temp(j,k+idproc,k)
           end do

        end do

      end do
       else

       do kw = 1, nvp
         blocklengths(kw) = kxp
       end do
      call MPI_REDUCE_SCATTER(wktemp, f, blocklengths,
mcplx, MPI_SUM, lgrp, ierr)

       end if

!      print *, 'Initializing FFT'

!      isign = 0
      kstrt = 1+idproc
!      call pfft1c(f,g,isign,mixup,sct,indx,kstrt,kxp,kblok)
!      print *, kxp, kblok
!      print *, mixup
!      print *, sct
      print *, 'Deposit: Calling FFT'
      isign = 1
      call pfft1c(f,g,isign,mixup,sct,indx,kstrt,kxp,kblok)

      do k = 1, kblok
        wfcn(1,l,k) = 0
        do j = 1, kxp
!          if (.false.) then !(j-(j/2)*2).ne.0) then      !
I'd love a "if (j&1)" right here
```

```
!            wfcn(j+1,l,k) = - f(j,k)
!          else
             wfcn(j+1,l,k) = f(j,k)
!          end if
         end do
         do j = kxp+2, nxpmx
           wfcn(j,l,k) = 0
         end do
       end do

       end if


      if (.false.) then  ! make sure wfcn periodic by adding
the right a*(x-(nx/2))

       if (.false.) then ! shared memory
         ctemp = ( wfcn(2,l,1) - wfcn(1+nxp(nblok),l,nblok)
) / (nx - 1)
       end if

       if (.true.) then ! distributed mpi

         if (idproc.eq.0) then ! recieve last w
           call MPI_IRECV(wtemp(2,1,1),1,mcplx,nvp-
1,nvp,lgrp,msid,ierr)
         end if
         if (idproc.eq.(nvp-1)) then ! send last w
           call
MPI_SEND(wfcn(1+nxp(nblok),l,nblok),1,mcplx,0,nvp,lgrp,ierr)
         end if

         if (idproc.eq.0) then
           call MPI_WAIT(msid,istatus,ierr)
           ctemp = ( wfcn(2,l,1) - wtemp(2,1,1) ) / (nx - 1)
! calculate coefficient and tell it to everyone else
           do k = 2, nvp
             call MPI_SEND(ctemp,1,mcplx,k-1,0,lgrp,ierr)
           end do
         else
! get coefficient from proc 0
           call MPI_IRECV(ctemp,1,mcplx,0,0,lgrp,msid,ierr)
           call MPI_WAIT(msid,istatus,ierr)
         end if

       end if

       do k = 1, nblok
         dx = noff(k) - 2 - 0.5*nx
```

```
          do j = 2, nxp(k)+1
            wfcn(j,l,k) = wfcn(j,l,k) + ctemp * (j + dx)
          end do

        end do

      end if

      if (.true.) then  ! zero edges

        if (idproc.eq.0) then
          wfcn(1,l,1) = cmplx(0.,0.)
          wfcn(2,l,1) = cmplx(0.,0.)
          wfcn(3,l,1) = cmplx(0.,0.)
          wfcn(4,l,1) = cmplx(0.,0.)
         if (.true.) then
          wfcn(5,l,1) = cmplx(0.,0.)
          wfcn(6,l,1) = cmplx(0.,0.)
         if (.false.) then
          wfcn(7,l,1) = cmplx(0.,0.)
          wfcn(8,l,1) = cmplx(0.,0.)
          wfcn(9,l,1) = cmplx(0.,0.)
          wfcn(10,l,1) = cmplx(0.,0.)
!          wfcn(11,l,1) = cmplx(0.,0.)
         end if
         end if
        end if
        if (idproc.eq.(nvp-1)) then
         if (.true.) then
         if (.false.) then
!          wfcn(-7+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-6+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-5+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-4+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-3+nxp(nblok),l,nblok) = cmplx(0.,0.)
         end if
          wfcn(-2+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(-1+nxp(nblok),l,nblok) = cmplx(0.,0.)
         end if
          wfcn(0+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(1+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(2+nxp(nblok),l,nblok) = cmplx(0.,0.)
          wfcn(3+nxp(nblok),l,nblok) = cmplx(0.,0.)
        end if

      end if

      end subroutine
```

Listing E. Wavefunction reconstruction / virtual classical particle deposit routine.

X. Appendix C

- The source code for the visualization and the quantum data formats

A. Visualization

The following list codes used to visualize the output of the quantum PIC code. These listings are in C.

```
#define ColorReferenceVal     0xC000L           //=0.75*0x10000
      /*    =0.75*65536=0x0.C*0x10000
      This describes the brightness of the pixel      */

Handle phaseColorHandle=nil;
long *phaseColorArray=nil;
long Complex2PhaseColor(Complex *in, long *outPhaseOnlyColor, myReal
*magSqOut);
//Converts a Complex number into a phase color representation,
compacted into 0x00rrggbb
long Complex2PhaseColor(Complex *in, long *outPhaseOnlyColor, myReal
```

```
*magSqOut)
      {long out=0;
      if (!phaseColorHandle) {      long tli;
            phaseColorHandle=NewHandle(514L<<2); //Allocating room for
514 longs
            MoveHHi(phaseColorHandle); HLock(phaseColorHandle);
            phaseColorArray=(long*)*phaseColorHandle;
            for(tli=514; tli--; ) {
                  HSVColor tHSV={0,MaxSmallFract,MaxSmallFract};
                  RGBColor tRGB;
                  tHSV.hue=MaxSmallFract*((tli<256)?tli+256:tli-
256)>>9;
                  HSV2RGB(&tHSV,&tRGB);
                  phaseColorArray[tli]=((long)tRGB.red&0xff00)<<8
                        |     ((long)tRGB.green&0xff00)
                        |     ((long)tRGB.blue)>>8;
                  }
            phaseColorArray+=256;
            }
      if (phaseColorHandle) {
            myReal magSq=in->a*in->a;
            myReal p=CPhase((*in))*invPi; //Convert [-π,π] to [-1,1]
            long tli, br;
            magSq+=in->b*in->b;
            tli=p*256;
            br=magSq*ColorReferenceVal;
            out=phaseColorArray[tli];
            if (magSqOut) *magSqOut=magSq;
            if (outPhaseOnlyColor) *outPhaseOnlyColor=out;

            tli=br*((Byte*)&out)[1];
            ((Byte*)&out)[1]=(tli&0xff000000)?0xff:(tli>>16);
            tli=br*((Byte*)&out)[2];
            ((Byte*)&out)[2]=(tli&0xff000000)?0xff:(tli>>16);
            tli=br*((Byte*)&out)[3];
            ((Byte*)&out)[3]=(tli&0xff000000)?0xff:(tli>>16);

            }

      return out;
      }

void DrawPhaseCircle(short left, short top)
      {     long tx,ty; Complex tC; RGBColor tRGB; myReal tr; long
tli;
      tr=0.9*PhaseCircleRadius; tr=1/tr;
      for(ty=0; ty<(PhaseCircleRadius<<1); ty++)
            for(tx=0; tx<(PhaseCircleRadius<<1); tx++)
                  if ((tx-PhaseCircleRadius)*(tx-
PhaseCircleRadius)+(ty-PhaseCircleRadius)*(ty-
PhaseCircleRadius)<PhaseCircleRadius*PhaseCircleRadius) {
                        tC.a=tr*(tx-PhaseCircleRadius);
                        tC.b=tr*(PhaseCircleRadius-ty); //Because up
is the negative y direction in graphics
```

```
                        //ComplexSq2RGB(&tC, &tRGB, nil, true);
                        tli=Complex2PhaseColor(&tC, nil, nil);
                        tRGB.red=0xff00&(tli>>8);
                        tRGB.green=0xff00&(tli);
                        tRGB.blue=(0xff&tli)<<8;
                        RGBForeColor(&tRGB);
                        MoveTo(tx+left,ty+top); Line(0,0);
                        }
      }


void DisplayWavefcn(Complex *wf, long sizeX, long bandOffset, long
bandSize, long flags)
      {     long baseAddr=(long)(*mainGWPM)->baseAddr,
                  rowBytes=0x3fff&(long)(*mainGWPM)->rowBytes,
tx,ty;//, sizeX=1<<lgArenaSizeX;
      long tmpTC=TickCount(), lastheight=DisplayHeight-1;
      Rect tRect={0,0, DisplayHeight, 1<<lgArenaSizeX};
      RGBColor tRGB={0,0,0};
      tRect.right=sizeX;

      ForeColor(whiteColor); BackColor(blackColor);
      if (!(flags&DontErase)) EraseRect(&tRect);

      for(tx=sizeX; tx--; ) {
            long colorPhaseMag, colorPhase;
            myReal magSq;

            ForeColor(whiteColor); MoveTo(tx, DisplayHeight-1);
Line(0,0);
            /*ComplexSq2RGB(wfptr+tx, &tRGB, nil, true); //or
&wfptr[tx] pg 99 of K&R
            tp[tx]=((long)tRGB.red&0xff00)<<8
                        |     ((long)tRGB.green&0xff00)
                        |     ((long)tRGB.blue)>>8; */
            {Complex tC=wf[tx];
      //    CScalar(tC, tC, 4);
            colorPhaseMag=Complex2PhaseColor(&tC/*wf+tx*/,
&colorPhase, &magSq);
            }
            if (flags&PlotWhite) magSq=wf[tx].a;
            else if (flags&DontDrawBand) magSq=wf[tx].a+wf[tx].b;
            else {
                  long tli;
#define cmultiplier                 8
                  tli=cmultiplier*((colorPhaseMag&0xff0000)>>16);
                  tRGB.red=tli>0x00ffL?0xffffL:0x0101L*tli;
                  tli=cmultiplier*((colorPhaseMag&0xff00)>>8);
                  tRGB.green=tli>0x00ffL?0xffffL:0x0101L*tli;
                  tli=cmultiplier*(colorPhaseMag&0xff);
                  tRGB.blue=tli>0x00ffL?0xffffL:0x0101L*tli;
#undef cmultiplier
      /*    tRGB.red=0x0101L*((colorPhaseMag&0xff0000)>>16); //0x0101L
            tRGB.green=0x0101L*((colorPhaseMag&0xff00)>>8);
```

```
            tRGB.blue=0x0101L*(colorPhaseMag&0xff);*/
            if (!(flags&PlotWhite)) RGBForeColor(&tRGB);
            MoveTo(tx, DisplayHeight-bandSize-1+bandOffset);
Line(0,bandSize);
            }

            tRGB.red=0x0101L*((colorPhase&0xff0000)>>16);
            tRGB.green=0x0101L*((colorPhase&0xff00)>>8);
            tRGB.blue=0x0101L*(colorPhase&0xff);
            if (!(flags&PlotWhite)) RGBForeColor(&tRGB);
            if (flags&ConnectDots) {
                  MoveTo(tx+1, lastheight);
                  LineTo(tx,
lastheight=(flags&DontDrawBand?bandOffset:DisplayHeight-1)-
magSq*DisplayHeight);
                  }
            else {
                  MoveTo(tx,
lastheight=(flags&DontDrawBand?bandOffset:DisplayHeight-1)-
magSq*DisplayHeight); Line(0,0); //  *0.75
                  }
            }


      tmpTC=TickCount()-tmpTC;

tmpTC=0x03303030+(tmpTC%10)+(((tmpTC/10)%10)<<8)+(((tmpTC/100)%10)<<16
);
      //MoveTo(mainGWRect.right, 16); ForeColor(whiteColor);
DrawString((StringPtr)&tmpTC);
      }
```

Listing F. Code used to visualize the data from the quantum PIC code.

B. Quantum Correlation Analysis, Eigenstate Extraction, and Data Reader

Most of the correlation analysis of the quantum was a custom-built parallel code, with its own ability to read binary data files output by the quantum PIC code. This listing is in C. These routines are presented together because of their interdependence.

```
void TransposeInMNOutData(long *inP, long *outP, long lgElemSize /*in
longs*/, long M, long N);
void TransposeInMNOutData(long *inP, long *outP, long lgElemSize /*in
longs*/, long M, long N)
      {/* Transpose a M elements/row, N row matrix in inP to outP */
      if (inP) if (outP)
      if (lgElemSize>=0)
      if (M>=1)
      if (N>=1) {
            long j=N;

            while (j--) {
                  long i=M;

                  while (i--) {
                        long k=1L<<lgElemSize;
                        while (k--) {
                              long
tli=inP[k+(i<<lgElemSize)+(j*M<<lgElemSize)];

outP[k+(j<<lgElemSize)+(i*N<<lgElemSize)]=tli;
                              }

                        }

                  }

            }

      }

void TransposeSquareData(long *inP, long elemSize /*in longs*/, long
M);
void TransposeSquareData(long *inP, long elemSize /*in longs*/, long
M)
      {
      if (inP)
      if (elemSize>0)
      if (M>1) {
            long j=M;

            while (--j) {
                  long i=j;

                  while (i--) {
                        long k=elemSize;
                  //    printf("Swapping: (%d,%d)\n", i, j);
                        while (k--) {
                              long tli=inP[k+i*elemSize+j*M*elemSize];
                              long
tli2=inP[k+j*elemSize+i*M*elemSize];
                              inP[k+j*elemSize+i*M*elemSize]=tli;
                              inP[k+i*elemSize+j*M*elemSize]=tli2;
                              }
```

```
                        }

                  }

            }

      }

#define Dim       10
void TestTransposers()
      {
      long array[Dim][Dim];
      long array2[Dim*Dim];
      printf("Testing Square Transposer\n");

      {long j, count=0;
      for(j=0; j<Dim; j++)
            {long i;
            for(i=0; i<Dim; i++) {
                  array[j][i]=count++;
                  printf("%6d", array[j][i]);
                  }
            printf("\n");
            }
      }

      TransposeSquareData((long*)array, 1, Dim);

      {long j;
      for(j=0; j<Dim; j++)
            {long i;
            for(i=0; i<Dim; i++) {
                  printf("%6d", array[j][i]);
                  }
            printf("\n");
            }
      }

      printf("Testing Regular Transposer\n");

      TransposeInMNOutData((long*)array, array2, 0, Dim, Dim-2);

      {long j;
      for(j=0; j<Dim; j++)
            {long i;
            for(i=0; i<Dim-2; i++) {
                  printf("%6d", array2[j*(Dim-2)+i]);
                  }
            printf("\n");
            }
      }
```

```
	}


BabyQFileInfoStruct bfi[MaxBabyQFiles];


short CheckIndexOkay(short in);
short CheckIndexOkay(short in)
	{short out=0;
	if ((in>0)&&(in<=MaxBabyQFiles))
		if (bfi[in].status)
			out=1;
	return out;
	}

long GetNumQ(short in);
long GetNumQ(short in)
	{long out=0;
	if (CheckIndexOkay(in))
		out=bfi[in].numQ;
	return out;
	}

long GetNumSteps(short in);
long GetNumSteps(short in)
	{long out=0;
	if (CheckIndexOkay(in))
		out=bfi[in].numSteps;
	return out;
	}


void DisposeBabyQDH(short in);
void DisposeBabyQDH(short in)
	{
	if (CheckIndexOkay(in))
		if (bfi[in].dataHandle) {
			HUnlock(bfi[in].dataHandle);
			DisposeHandle(bfi[in].dataHandle);
			bfi[in].dataHandle=nil;
			bfi[in].dataP=nil;
			bfi[in].dataMode=0;
			bfi[in].dataMemSize=0;
			}

	}

void CloseBabyQBinFile(short in, long idproc, long nproc)
	{//Close file and release related memory
	if (bfi[in].status) {
		bfi[in].status=0;
		DisposeBabyQDH(in);

		if (!idproc)
```

```
                  fclose(bfi[in].fp);
            bfi[in].fp=nil;

            {long i=sizeof(BabyQFileInfoStruct)>>2;
            while (i--) ((long*)&bfi[in])[i]=0;
            }

            }
      }

short OpenBabyQBinFile(const char *inFileName, long idproc, long
nproc)
      {short out=0;
if (!idproc)
      if (inFileName) {
            {long i;
            for(i=1; (i<=MaxBabyQFiles)&&!out; i++)
                  if (!bfi[i].status) out=i;
            }
            if (out) {

                  printf("Opening…");
/*                if (noErr==FSpOpenDF(inFS, fsRdPerm,
&bfi[out].fref)) */
                  bfi[out].fp=fopen(inFileName, "rb");
                  if (bfi[out].fp) {
                        BabyQBinaryHeaderStruct header;
                        long byteCount;
                        short validHeader=0; OSErr err;
                  //    GetEOF(bfi[out].fref, &bfi[out].fileSize);
                        fseek(bfi[out].fp, 0, SEEK_END);
                        bfi[out].fileSize=ftell(bfi[out].fp);
                  //    SetFPos(bfi[out].fref, fsFromStart,
bfi[out].currentPosition=0);
                        fseek(bfi[out].fp, bfi[out].currentPosition=0,
SEEK_SET);
                        bfi[out].status=1;

                        byteCount=sizeof(BabyQBinaryHeaderStruct);
                        printf("Reading Header…");
                  //    err=FSRead(bfi[out].fref, &byteCount,
&header); //Read header
                        byteCount=fread(&header, 1L, byteCount,
bfi[out].fp); //Read header

                        if (bfi[out].fileSize>64)
                        if (byteCount>=32)
                        if (!~(header.negVersion|0x0000ffffL)) {//top
16 bits are all 1's
                              if
((sizeof(float)==header.floatSize)||(sizeof(double)==header.floatSize)
)
                              if (header.dataOffset>=24) {
```

```
bfi[out].currentPosition+=header.dataOffset;
                              //    SetFPos(bfi[out].fref,
fsFromStart, bfi[out].currentPosition);
                                    fseek(bfi[out].fp,
bfi[out].currentPosition=0, SEEK_SET);
                                    validHeader=1;
                                    }
                              }
                        else
/*validHeader=EnterManualHeader(&header)?1:0*/;

                        if (validHeader) {
#define Xfer(e)         bfi[out].e=header.e
                              Xfer(dataOffset);
                              Xfer(floatSize);
                              Xfer(numSteps);
                              Xfer(numQ);
                              Xfer(sizeX);
                              Xfer(nvp);
                              Xfer(numPreGridCells);
                              Xfer(numPostGridCells);
#undef Xfer


bfi[out].dataFrameSize=bfi[out].floatSize*2*bfi[out].sizeX

+bfi[out].nvp*(bfi[out].numPreGridCells+bfi[out].numPostGridCells);

bfi[out].timeFrameSize=bfi[out].dataFrameSize*bfi[out].numQ;

                        //Readjust numSteps in case the file size is
short
                        {unsigned long altNumSteps=(bfi[out].fileSize-
bfi[out].dataOffset)/
                              (bfi[out].timeFrameSize);
                        if (altNumSteps<bfi[out].numSteps)
bfi[out].numSteps=altNumSteps;
                        }
                        //Readjust numSteps to power of 2
                        {unsigned long lgNumSteps, altNumSteps;
                        for(lgNumSteps=14/*/30/**/;
bfi[out].numSteps<(1L<<lgNumSteps); lgNumSteps--) ;

                        altNumSteps=1L<<lgNumSteps;
                        if (altNumSteps<bfi[out].numSteps)
bfi[out].numSteps=altNumSteps;
                        }


bfi[out].fileSize=bfi[out].dataOffset+bfi[out].numSteps*
                              (bfi[out].timeFrameSize);

                              }
                        else {
```

```
                              printf("Incorrect data format.\n");
                              CloseBabyQBinFile(out, idproc, nproc);
                              out=0;
                        }

                        }
                  else out=0;
                  }
            }
      MPI_Bcast(&out, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Bcast(&bfi[out], sizeof(BabyQFileInfoStruct), MPI_BYTE, 0,
MPI_COMM_WORLD);
      return out;
      }

inline long StartIndex(long idproc, long nproc, long ySize);
inline long StartIndex(long idproc, long nproc, long ySize)
      {
      return (nproc?((ySize)*idproc/nproc):idproc?(ySize):0);
      }
//#define StartIndex(idproc, nproc, ySize)
(nproc?((long)(ySize)*idproc/nproc):idproc?(ySize):0)

inline long NumIndicies(long idproc, long nproc, long ySize);
inline long NumIndicies(long idproc, long nproc, long ySize)
      {
      return (StartIndex(idproc+1, nproc, ySize)
            -StartIndex(idproc, nproc, ySize));
      }

short LoadAllBabyQData(short in, long idproc, long nproc)
      {short out=0;
      if (CheckIndexOkay(in)) {
                  long maxIndex, elemSize, storageAreaBytes,
processAreaBytes,
                        swapSpaceBytes, mpiInfoBytes, requestedBytes;
                  DisposeBabyQDH(in);


maxIndex=bfi[in].dataFrameSize/(elemSize=2*bfi[in].floatSize);

                              storageAreaBytes=elemSize*bfi[in].numQ*

bfi[in].numSteps*(NumIndicies(idproc, nproc, maxIndex));


processAreaBytes=max(sizeof(Complex)*2,elemSize*bfi[in].numQ)*

bfi[in].numSteps*(NumIndicies(idproc, nproc, maxIndex));


swapSpaceBytes=idproc?(1L<<21):(nproc<<21); //Try two megs per node
                              if (swapSpaceBytes>(64L<<20))
                                    swapSpaceBytes=64L<<20; //but no
```

```
more than 64 megs
                              //Round down

swapSpaceBytes=(swapSpaceBytes/bfi[in].timeFrameSize)*bfi[in].timeFram
eSize;


mpiInfoBytes=(sizeof(MPI_Request)+sizeof(long))*nproc;


bfi[in].dataHandle=NewHandle(requestedBytes=storageAreaBytes

+max(processAreaBytes,swapSpaceBytes)
                              +mpiInfoBytes);
                        if (bfi[in].dataHandle) {
                              MoveHHi(bfi[in].dataHandle);
                              HLock(bfi[in].dataHandle);

bfi[in].dataP=(Byte*)*bfi[in].dataHandle; //Get pointer
                              {long *tp=(long*)bfi[in].dataP,
i=requestedBytes>>2;
                              while(i--) tp[i]=0; }   //Clear
memory

                              out=1;
                              }
                        else
                              out=0; /*printf("Low memory
mode…\n")*/

                        if (idproc) {
                              long info[2];
                              info[0]=out;
                              info[1]=requestedBytes;
                              MPI_Send(info, 2, MPI_INT, 0,
idproc, MPI_COMM_WORLD);
                              }
                        else {
                              long i;

                              for(i=nproc; i--; )
                                    if (i) {
                                          long info[2];
                                          MPI_Status stat;
                                          MPI_Recv(info, 2,
MPI_INT, i, i, MPI_COMM_WORLD, &stat);
                                          if (!info[0]) {
                                                out=0;
                                                printf("Node #%d
unable to allocate %d bytes.\n", i, info[1]);
                                                }
                                          }
                                    else {
                                          if (!bfi[in].dataP) {
```

```
                                                      out=0;
                                                      printf("Node #0
unable to allocate %d bytes.\n", requestedBytes);
                                                      }
                                                }

                                    }

                              MPI_Bcast(&out, 1, MPI_INT, 0,
MPI_COMM_WORLD);

                              if (out) {
                                    //Proceed to read all data into
the cluster

                                    if (idproc) {
                                          long
recvAreaBytes=((requestedBytes-storageAreaBytes-mpiInfoBytes))&~7;
                                          Byte *procP=bfi[in].dataP;
                                          Byte
*recvP=procP+storageAreaBytes;
                                          MPI_Request
*mpiInfo=(MPI_Request *)(recvP+recvAreaBytes);
                                          long *mpiDoneFlags=(long
*)(mpiInfo+nproc);
                                          long totalStepsRead=0;
                                          long lgElemSize=0;
                                          short looping=1;
                                          //Assuming sizeX is evenly
divisible by nproc
                                          for(;
bfi[in].dataFrameSize/nproc>(4L<<lgElemSize); lgElemSize++) ;

                                          while (looping) {
                                                long tsRecv=0;
                                                MPI_Status stat;


MPI_Irecv(recvP,

recvAreaBytes,

MPI_BYTE, 0, idproc,

MPI_COMM_WORLD, mpiInfo);

                                                MPI_Wait(mpiInfo,
&stat);

tsRecv=stat.len/(NumIndicies(idproc, nproc, maxIndex)
                                                      *bfi[in].numQ

*bfi[in].floatSize*2);
```

```
                                                printf("%d bytes
received. Transposing… ", stat.len);

                                                //Transpose and move
the data

TransposeInMNOutData((long*)recvP,

(long*)(procP+(totalStepsRead

*NumIndicies(idproc, nproc, maxIndex)

*bfi[in].numQ

*bfi[in].floatSize*2)),
                                                      lgElemSize,
tsRecv*bfi[in].numQ, 1);


totalStepsRead+=tsRecv;

                                                printf("%d steps so
far.\n", totalStepsRead);

                                                if
(totalStepsRead>=bfi[in].numSteps)
                                                      looping=0;
                                                }

                                          bfi[in].dataMode=1;

bfi[in].dataMemSize=totalStepsRead
                                                *NumIndicies(idproc,
nproc, maxIndex)
                                                *bfi[in].numQ
                                                *bfi[in].floatSize*2;

                                          }
                                    else {// node 0
                                          long
readAreaBytes=((requestedBytes-storageAreaBytes-mpiInfoBytes)>>1)&~7;
                                          Byte *procP=bfi[in].dataP;
                                          Byte
*readP=procP+storageAreaBytes;
                                          Byte
*sendP=readP+readAreaBytes;
                                          MPI_Request
*mpiInfo=(MPI_Request *)(sendP+readAreaBytes);
                                          long *mpiDoneFlags=(long
*)(mpiInfo+nproc);
                                          long totalStepsRead=0;
                                          long lgElemSize=0;
                                          short looping=1;
                                          //Assuming sizeX is evenly
```

```
divisible by nproc
                                        for(;
bfi[in].dataFrameSize/nproc>(4L<<lgElemSize); lgElemSize++) ;

                                        while (looping) {
                                              long byteCount;

                                              if
(totalStepsRead<bfi[in].numSteps) {
                                                    printf("Reading
file…");

fseek(bfi[in].fp, bfi[in].currentPosition=

bfi[in].dataOffset+totalStepsRead*bfi[in].timeFrameSize,
                                                          SEEK_SET);

byteCount=readAreaBytes;

byteCount=fread(readP, 1, byteCount, bfi[in].fp);
                                                    }
                                              else byteCount=0;

#ifdef testdata
                              printf("Generating test data at node
0\n");
                              {long ts;
                              for
(ts=(byteCount/bfi[in].timeFrameSize); ts>=0; ts--) {
                                    long x=bfi[in].sizeX;
                                    ComplexSingle
*d1tsP=(ComplexSingle *)(readP+
                                          (ts*bfi[in].timeFrameSize));
                                    ComplexSingle
*d2tsP=d1tsP+bfi[in].sizeX;
                                    ComplexSingle tC;
                                    float tr;

tC.a=cos((Pi/32.0)*(totalStepsRead+ts));
                                    tC.b=sin((-
Pi/32.0)*(totalStepsRead+ts));
                                    while (x--) {
                                          float
snx=sin(x*(Pi/bfi[in].sizeX));
                                          float
sn2x=sin(x*(Pi/bfi[in].sizeX)*2);
                                          float
sn3x=sin(x*(Pi/bfi[in].sizeX)*3);
                                          ComplexSingle tC2, tC3,
psi2;
                                          CScalar(d1tsP[x],tC,snx);

                                          CMult(tC2, tC, tC);
                                          CScalar(psi2,tC2,sn2x);
```

```
                                        CMult(tC3,tC,tC2);
d2tsP[x].a=sn3x*tC3.a+psi2.a;
d2tsP[x].b=sn3x*tC3.b+psi2.b;
                                        }
                                    }
                                }
#endif
                                              if (totalStepsRead>0)
{
                                                    short waiting=1;

                                                    while (waiting)
{
                                                          short
finished=1;
                                                          long
lastWait=0;
                                                          long i;
                                                          for(i=1;
i<nproc; i++) {
                                                                int
flag;
MPI_Status stat;
                                                                if
(!mpiDoneFlags[i]) {
MPI_Test(mpiInfo+i, &flag, &stat);
if (!flag) {
      finished=0;
      if (lastWait!=i) {
            printf("Still waiting on node #%d…\n", i);
            lastWait=i;
            }
      else printf(".");
//    break;
      }
else
      mpiDoneFlags[i]=1;
```

```
}
                                                                    }
                                                                if
(finished) waiting=0;
                                                                }
                                                            }
                                                        //Now the send memory
is free
                                                        if (byteCount>0) {
                                                            long
tsRead=byteCount/bfi[in].timeFrameSize;
                                                            long i;
                                                            if
(tsRead+totalStepsRead>bfi[in].numSteps)
tsRead=bfi[in].numSteps-totalStepsRead;
                                                            printf(" %d
steps read.\n", tsRead);
printf("Transposing data for sending… ");
                                                            //Transpose the
data to make it easier to send
TransposeInMNOutData((long*)readP, (long*)sendP,
lgElemSize, nproc, tsRead*bfi[in].numQ);
printf("MPI_Isends… ");
                                                            //Head 'em up,
move 'em out
                                                            for(i=1;
i<nproc; i++) {
MPI_Isend(sendP+(StartIndex(i, nproc, maxIndex)*
tsRead*bfi[in].numQ*bfi[in].floatSize*2),
(NumIndicies(i, nproc, maxIndex)
*tsRead*bfi[in].numQ*bfi[in].floatSize*2),
MPI_BYTE, i, i,
MPI_COMM_WORLD, mpiInfo+i);
mpiDoneFlags[i]=0;
                                                                }
```

```
printf("Transposing data for node 0…\n");
                                                    //Transpose and
move the data that stays here
TransposeInMNOutData((long*)sendP,
(long*)(procP+(totalStepsRead
*bfi[in].numQ
<<(lgElemSize+2))),
lgElemSize, tsRead*bfi[in].numQ, 1);

totalStepsRead+=tsRead;
                                                          printf(" %d
steps so far.\n", totalStepsRead);
                                                          }
                                                    else looping=0;
                                                    }
                                              bfi[in].dataMode=1;
bfi[in].dataMemSize=totalStepsRead
                                                    *NumIndicies(idproc,
nproc, maxIndex)
                                                    *bfi[in].numQ
                                                    <<(lgElemSize+2);
                                              }
                                        printf("Yum!\n");
                                        /*The data should now be in a
format where space
                                        is divided among processors for
all q, but the time
                                        sequence for any one (q,x) is in
rows within a processor. */
                                        }
                                  else
                                        printf("Insufficient memory on
cluster.\n");
                  }
      return out;
      }
```

```
void RenormalizeBabyQData(short in, long spinType, long idproc, long
nproc);
void RenormalizeBabyQData(short in, long spinType, long idproc, long
nproc)
      {
      long maxIndex=bfi[in].dataFrameSize/(2*bfi[in].floatSize);
      if (bfi[in].numQ<2) spinType=maxwellian;
      switch (bfi[in].floatSize) {
            case sizeof(float):
                  {
                  long sumRow=bfi[in].numQ<<1;
                  ComplexSingle *dp=(ComplexSingle *)bfi[in].dataP;
                  ComplexSingle *sumP=dp+bfi[in].numSteps*bfi[in].numQ
                        *NumIndicies(idproc, nproc, maxIndex);
                  ComplexSingle *sum2P=sumP+bfi[in].numSteps*sumRow;

#ifdef testdata
                              printf("Generating test data\n");
                              {long ts=bfi[in].numSteps;
                              while (ts--) {
                                    long i=bfi[in].numQ
                                          *NumIndicies(idproc, nproc,
maxIndex);
                                    ComplexSingle *dtsP=dp+(ts*i);
                                    ComplexSingle tC;
                                    tC.a=cos((Pi/32.0)*ts);
                                    tC.b=sin((Pi/32.0)*ts);
                                    while (i--)
                                          dtsP[i]=tC;
                                    }
                              }
#endif

                  //Clear arrays
                  {long i=bfi[in].numSteps*sumRow<<1;
                  while (i--) sumP[i].b=sumP[i].a=0;
                  }

                  printf(" summing within processor… \n");
                  switch (spinType) {
                        case fermion:
                              {long ts=bfi[in].numSteps;
                              while(ts--) {
                                          {// Computing <Psi2|Psi1>
                                          ComplexSingle
*tp1=&dp[(ts*bfi[in].numQ)

*NumIndicies(idproc, nproc, maxIndex)];
                                          ComplexSingle
*tp2=&tp1[NumIndicies(idproc, nproc, maxIndex)];
                                          long x=NumIndicies(idproc,
nproc, maxIndex);
                                          float sumA=0, sumB=0;
                                          while (x--) {
```

```
sumA+=tp2[x].a*tp1[x].a+tp2[x].b*tp1[x].b;

sumB+=tp2[x].a*tp1[x].b-tp2[x].b*tp1[x].a;
                                                }

sumP[ts*sumRow+bfi[in].numQ].a=sumA;

sumP[ts*sumRow+bfi[in].numQ].b=sumB;
                                          }
                                    }

                              }
                        case maxwellian:
                        default:
                              {long ts=bfi[in].numSteps;
                              while(ts--) {
                                    long q=bfi[in].numQ;
                                    while (q--) {     //Computing
<PsiQ|PsiQ>
                                          ComplexSingle
*tp=&dp[(ts*bfi[in].numQ+q)

*NumIndicies(idproc, nproc, maxIndex)];
                                          long x=NumIndicies(idproc,
nproc, maxIndex);
                                          float sum=0;
                                          while (x--) {
                                                sum+=CMagSq(tp[x]);
                                                }
                                          sumP[ts*sumRow+q].a=sum;
                                    //    printf("|psi #%d|^2 is %f ",
q, sum);
                                          }
                                    }

                              }
                              break;
                        }

                  printf(" summing across processors… \n");
                  MPI_Allreduce(sumP, sum2P,
                        bfi[in].numSteps*sumRow*2, MPI_FLOAT, MPI_SUM,
MPI_COMM_WORLD);

                  printf(" Calculating normalization factors… \n");
                  switch (spinType) {
                        case fermion:
                              {long ts=bfi[in].numSteps;
                              while(ts--) {
                                    float
norm=sum2P[ts*sumRow+0].a*sum2P[ts*sumRow+1].a
                                                -
CMagSq(sum2P[ts*sumRow+bfi[in].numQ]);
```

```
                                    long q=bfi[in].numQ;

norm=norm>0?1.0/sqrt(sqrt(2.0*norm)):1.0;
                                    while (q--) {
                                          sumP[ts*sumRow+q].a=norm;
                                          }
                                    }
                              }
                              break;
                        case maxwellian:
                        default:
                              {long ts=bfi[in].numSteps;
                              while(ts--) {
                                    long q=bfi[in].numQ;
                                    while (q--) {

sumP[ts*sumRow+q].a=sum2P[ts*sumRow+q].a?

1.0/sqrt(sum2P[ts*sumRow+q].a):1.0;
                                          }
                                    }
                              }
                              break;
                        }

                  printf(" Normalizing… \n");
                              {long ts=bfi[in].numSteps;
                              while(ts--) {
                                    long q=bfi[in].numQ;
                                    while (q--) {
                                          float
norm=sumP[ts*sumRow+q].a;
                                          ComplexSingle
*tp=&dp[(ts*bfi[in].numQ+q)

*NumIndicies(idproc, nproc, maxIndex)];
                                          long x=NumIndicies(idproc,
nproc, maxIndex);
                                          while (x--) {
                                                CScalar(tp[x], tp[x],
norm);

                                                }
                                          }
                                    }
                              }

                  }
                  break;
            case sizeof(double):
                  break;
            default: printf("Error.  Invalid float size.\n");
                  break;
                  }
```

```
	}

void DoFFTYBabyQ(short in, long lgNumSteps, long idproc, long nproc)
	{
	if (CheckIndexOkay(in)) {
		long maxIndex=bfi[in].dataFrameSize/(2*bfi[in].floatSize);

		switch (bfi[in].floatSize) {
			case sizeof(float):
				{
				ComplexSingle *dp=(ComplexSingle
*)bfi[in].dataP;

				{//Clear second half of data
				float *tp=&dp[NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ<<lgNumSteps].a;
				long i=NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ<<(lgNumSteps+1);
				while (i--) tp[i]=0;
				}

				DoFFTYCSingleN(dp, lgNumSteps+1,
					NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ);

				}
				break;
			case sizeof(double):
				{
				ComplexDouble *dp=(ComplexDouble
*)bfi[in].dataP;

				{//Clear second half of data
				double *tp=&dp[NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ<<lgNumSteps].a;
				long i=NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ<<(lgNumSteps+1);
				while (i--) tp[i]=0;
				}

				DoFFTYCDoubleN(dp, lgNumSteps+1,
					NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ);

				}
				break;
			default:
				break;
				}



		}
	}
```

```
short CorrelElemBabyQ(short in, long ts1, long ts2, Complex *outP,
long spinType, Complex *cxP, long idproc, long nproc)
      {short out=0;
      if (outP)
      if (CheckIndexOkay(in))
      if (ts1>=0) if (ts2>=0)
      if (ts1<bfi[in].numSteps)
      if (ts2<bfi[in].numSteps)
            {
            Byte *rawWF1P=nil, *rawWF2P=nil;
            long maxIndex=bfi[in].dataFrameSize/(2*bfi[in].floatSize);
            long myStI=StartIndex(idproc, nproc, maxIndex);
            long myNumI=NumIndicies(idproc, nproc, maxIndex);

            if (bfi[in].numQ<2) spinType=maxwellian;

            if (1==bfi[in].dataMode) {
                  if (bfi[in].dataP) {

rawWF1P=bfi[in].dataP+myNumI*(2*bfi[in].floatSize)*ts1;

rawWF2P=bfi[in].dataP+myNumI*(2*bfi[in].floatSize)*ts2;
                        }
                  }
            else {
                  }

            if (rawWF1P&&rawWF2P) {
                  long nxp, nxpmx,

numGrids=bfi[in].sizeX+bfi[in].nvp*(bfi[in].numPreGridCells+bfi[in].nu
mPostGridCells);
                  nxp=bfi[in].nvp?bfi[in].sizeX/bfi[in].nvp:0;

nxpmx=nxp+bfi[in].numPreGridCells+bfi[in].numPostGridCells;


                                                switch
(bfi[in].floatSize) {
                                                      case
sizeof(float):
      switch (spinType) {
            case fermion:
      {float *wpA, *wpB, a=0, b=0, c=0, d=0, normA, normB;
      Complex outCx;
      long x=numGrids;
      wpA=(float*)(rawWF1P);
      wpB=(float*)(rawWF2P);

      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
```

```
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+(x<<1)]*wpB[0+(x<<1)]+wpA[1+(x<<1)]*wpB[1+(x<<1)];
                  b=b+wpB[1+(x<<1)]*wpA[0+(x<<1)]-
wpA[1+(x<<1)]*wpB[0+(x<<1)];


c=c+wpA[0+(x<<1)+numGrids]*wpB[0+(x<<1)+numGrids]+wpA[1+(x<<1)+numGrid
s]*wpB[1+(x<<1)+numGrids];
                  d=d+wpB[1+(x<<1)+numGrids]*wpA[0+(x<<1)+numGrids]-
wpA[1+(x<<1)+numGrids]*wpB[0+(x<<1)+numGrids];
                  }
            }
      outCx.a=a*c-b*d;
      outCx.b=b*c+a*d;

      x=numGrids;
      a=0;
      b=0;
      c=0;
      d=0;
      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+(x<<1)]*wpB[0+(x<<1)+numGrids]+wpA[1+(x<<1)]*wpB[1+(x<<1)+nu
mGrids];
                  b=b+wpB[1+(x<<1)+numGrids]*wpA[0+(x<<1)]-
wpA[1+(x<<1)]*wpB[0+(x<<1)+numGrids];


c=c+wpA[0+(x<<1)+numGrids]*wpB[0+(x<<1)]+wpA[1+(x<<1)+numGrids]*wpB[1+
(x<<1)];
                  d=d+wpB[1+(x<<1)]*wpA[0+(x<<1)+numGrids]-
wpA[1+(x<<1)+numGrids]*wpB[0+(x<<1)];
                  }
            }
      outCx.a-=a*c-b*d;
      outCx.b-=b*c+a*d;

      x=numGrids;
      a=0;
      b=0;
      c=0;
      d=0;
      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+(x<<1)]*wpA[0+(x<<1)]+wpA[1+(x<<1)]*wpA[1+(x<<1)];
```

```
b=b+wpA[0+(x<<1)+numGrids]*wpA[0+(x<<1)+numGrids]+wpA[1+(x<<1)+numGrid
s]*wpA[1+(x<<1)+numGrids];


c=c+wpB[0+(x<<1)]*wpB[0+(x<<1)]+wpB[1+(x<<1)]*wpB[1+(x<<1)];

d=d+wpB[0+(x<<1)+numGrids]*wpB[0+(x<<1)+numGrids]+wpB[1+(x<<1)+numGrid
s]*wpB[1+(x<<1)+numGrids];
                  }
            }
      normA=a*b;
      normB=c*d;

      x=numGrids;
      a=0;
      b=0;
      c=0;
      d=0;
      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+(x<<1)]*wpA[0+(x<<1)+numGrids]+wpA[1+(x<<1)]*wpA[1+(x<<1)+nu
mGrids];
                  b=b+wpA[0+(x<<1)]*wpA[1+(x<<1)+numGrids]-
wpA[1+(x<<1)]*wpA[0+(x<<1)+numGrids];


c=c+wpB[0+(x<<1)]*wpB[0+(x<<1)+numGrids]+wpB[1+(x<<1)]*wpB[1+(x<<1)+nu
mGrids];
                  d=d+wpB[0+(x<<1)]*wpB[1+(x<<1)+numGrids]-
wpB[1+(x<<1)]*wpB[0+(x<<1)+numGrids];
                  }
            }
      normA-=a*a+b*b;
      normB-=c*c+d*d;

      normA=normA*normB;
      if (normA>0) normA=1.0/sqrt(normA);
      else normA=0;

      CAccumSMult(outP[0], outCx, normA);
      }
                  break;
            case boson:
                  break;
            case maxwellian:
            default:
      {

/*    for(index=0; index<myNumI; index++) {
```

```
            long x=(index+myStI)%numGrids, q=(index+myStI)/numGrids;

            }*/
      long q;
      for(q=myStI/numGrids; q<=(myStI+myNumI)/numGrids; q++) {
            long x;
            float *wpA=(float*)(rawWF1P)+((q*numGrids-myStI)<<1),
                  *wpB=(float*)(rawWF2P)+((q*numGrids-myStI)<<1);
            float a=0, b=0;

            for(x=myStI-q*numGrids<0?0:myStI-q*numGrids;
                        (x<numGrids)&&(x<myStI+myNumI-q*numGrids);
                        x++) {

                  if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):true)
                        {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+x<<1]*wpB[0+x<<1]+wpA[1+(x<<1)]*wpB[1+(x<<1)];
                        b=b+wpB[1+(x<<1)]*wpA[0+x<<1]-
wpA[1+(x<<1)]*wpB[0+x<<1];
                        }

                  }

            cxP[q].a=outP[q].a+a;
            cxP[q].b=outP[q].b+b;

            }

      }

      MPI_Allreduce(cxP, outP, bfi[in].numQ<<1,
                        sizeof(myReal)==4?MPI_FLOAT:MPI_DOUBLE,
                        MPI_SUM, MPI_COMM_WORLD);

/*    {long q=bfi[in].numQ;
      while (q--) {
      {float *wpA, *wpB, a=0, b=0;
      long x=numGrids;
      wpA=(float*)(rawWF1P+bfi[in].dataFrameSize*q);
      wpB=(float*)(rawWF2P+bfi[in].dataFrameSize*q);

      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+x<<1]*wpB[0+x<<1]+wpA[1+(x<<1)]*wpB[1+(x<<1)];
                  b=b+wpB[1+(x<<1)]*wpA[0+x<<1]-
wpA[1+(x<<1)]*wpB[0+x<<1];
                  }
```

```
            }
      outP[q].a+=a;
      outP[q].b+=b;
      }
            }
      }*/
                  break;
            }

                                                                  break;
                                                            case
sizeof(double):
      switch (spinType) {
            case fermion:
      {double *wpA, *wpB, a=0, b=0, c=0, d=0, normA, normB;
      Complex outCx;
      long x=numGrids;
      wpA=(double*)(rawWF1P);
      wpB=(double*)(rawWF2P);

      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+(x<<1)]*wpB[0+(x<<1)]+wpA[1+(x<<1)]*wpB[1+(x<<1)];
                  b=b+wpB[1+(x<<1)]*wpA[0+(x<<1)]-
wpA[1+(x<<1)]*wpB[0+(x<<1)];


c=c+wpA[0+(x<<1)+numGrids]*wpB[0+(x<<1)+numGrids]+wpA[1+(x<<1)+numGrid
s]*wpB[1+(x<<1)+numGrids];
                  d=d+wpB[1+(x<<1)+numGrids]*wpA[0+(x<<1)+numGrids]-
wpA[1+(x<<1)+numGrids]*wpB[0+(x<<1)+numGrids];
                  }
            }
      outCx.a=a*c-b*d;
      outCx.b=b*c+a*d;

      x=numGrids;
      a=0;
      b=0;
      c=0;
      d=0;
      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+(x<<1)]*wpB[0+(x<<1)+numGrids]+wpA[1+(x<<1)]*wpB[1+(x<<1)+nu
mGrids];
                  b=b+wpB[1+(x<<1)+numGrids]*wpA[0+(x<<1)]-
```

```
wpA[1+(x<<1)]*wpB[0+(x<<1)+numGrids];


c=c+wpA[0+(x<<1)+numGrids]*wpB[0+(x<<1)]+wpA[1+(x<<1)+numGrids]*wpB[1+
(x<<1)];
                  d=d+wpB[1+(x<<1)]*wpA[0+(x<<1)+numGrids]-
wpA[1+(x<<1)+numGrids]*wpB[0+(x<<1)];
                  }
            }
      outCx.a-=a*c-b*d;
      outCx.b-=b*c+a*d;

      x=numGrids;
      a=0;
      b=0;
      c=0;
      d=0;
      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+(x<<1)]*wpA[0+(x<<1)]+wpA[1+(x<<1)]*wpA[1+(x<<1)];

b=b+wpA[0+(x<<1)+numGrids]*wpA[0+(x<<1)+numGrids]+wpA[1+(x<<1)+numGrid
s]*wpA[1+(x<<1)+numGrids];


c=c+wpB[0+(x<<1)]*wpB[0+(x<<1)]+wpB[1+(x<<1)]*wpB[1+(x<<1)];

d=d+wpB[0+(x<<1)+numGrids]*wpB[0+(x<<1)+numGrids]+wpB[1+(x<<1)+numGrid
s]*wpB[1+(x<<1)+numGrids];
                  }
            }
      normA=a*b;
      normB=c*d;

      x=numGrids;
      a=0;
      b=0;
      c=0;
      d=0;
      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wpA[0+(x<<1)]*wpA[0+(x<<1)+numGrids]+wpA[1+(x<<1)]*wpA[1+(x<<1)+nu
mGrids];
                  b=b+wpA[0+(x<<1)]*wpA[1+(x<<1)+numGrids]-
wpA[1+(x<<1)]*wpA[0+(x<<1)+numGrids];
```

```
c=c+wpB[0+(x<<1)]*wpB[0+(x<<1)+numGrids]+wpB[1+(x<<1)]*wpB[1+(x<<1)+nu
mGrids];
                  d=d+wpB[0+(x<<1)]*wpB[1+(x<<1)+numGrids]-
wpB[1+(x<<1)]*wpB[0+(x<<1)+numGrids];
                  }
            }
      normA-=a*a+b*b;
      normB-=c*c+d*d;

      normA=normA*normB;
      if (normA>0) normA=1.0/sqrt(normA);
      else normA=0;

      CAccumSMult(outP[0], outCx, normA);
      }
                  break;
            case boson:
                  break;
            case maxwellian:
            default:
      {long q=bfi[in].numQ;
      while (q--) {
      {double *wp1, *wp2, a=0, b=0;
      long x=numGrids;
      wp1=(double*)(rawWF1P+bfi[in].dataFrameSize*q);
      wp2=(double*)(rawWF2P+bfi[in].dataFrameSize*q);

      while (x--) {
            if
(nxp?((x%nxpmx>=bfi[in].numPreGridCells)&&(x%nxpmx<nxp+bfi[in].numPreG
ridCells)):1)
                  {//∫ <psi(x,t+tau)|psi(x,t)> dx

a=a+wp1[0+x<<1]*wp2[0+x<<1]+wp1[1+(x<<1)]*wp2[1+(x<<1)];
                  b=b+wp2[1+(x<<1)]*wp1[0+x<<1]-
wp1[1+(x<<1)]*wp2[0+x<<1];
                  }
            }
      outP[q].a+=a;
      outP[q].b+=b;
      }
            }
      }
                  break;
            }
                                                      break;
                                                default:
                                                      break;
                                                }

                  }
            }
      return out;
```

```
	}

#define MaxEigenMegs		32L
void CorrBabyQSpin(short in, long lgNumSteps, long spinType, Complex
*cxOutP, long cxOutRow, FILE *tempEigenFile, long idproc, long nproc);
void CorrBabyQSpin(short in, long lgNumSteps, long spinType, Complex
*cxOutP, long cxOutRow, FILE *tempEigenFile, long idproc, long nproc)
	{
	long maxIndex=bfi[in].dataFrameSize/(2*bfi[in].floatSize);
	if (bfi[in].numQ<2) spinType=maxwellian;
	if (cxOutP) if (cxOutRow) {
		{long i=bfi[in].numSteps*cxOutRow; //Clear output
		while (i--) cxOutP[i].b=cxOutP[i].a=0;
		}

	switch (spinType) {
		case fermion:
			switch (bfi[in].floatSize) {
				case sizeof(float):
					{
					ComplexSingle *dp=(ComplexSingle
*)bfi[in].dataP;
					Complex
*p12P=(Complex*)(dp+bfi[in].numSteps*bfi[in].numQ
							*NumIndicies(idproc, nproc,
maxIndex));
					ComplexSingle *p1x2P=(ComplexSingle
*)(p12P+bfi[in].numSteps
							*NumIndicies(idproc, nproc,
maxIndex));
					ComplexSingle
*p2x2P=p1x2P+bfi[in].numSteps;
				//	ComplexSingle
*scratchP=p2x2P+bfi[in].numSteps;
					double startTime=MPI_Wtime();
					long x2;

					//Clear arrays
					{long i=bfi[in].numSteps
							*NumIndicies(idproc, nproc,
maxIndex);
					while (i--) p12P[i].b=p12P[i].a=0;
					}
#ifdef testdata
					printf("Generating test data\n");
					{long ts=bfi[in].numSteps;
					while (ts--) {
						long x=NumIndicies(idproc, nproc,
maxIndex);
						ComplexSingle
*d1tsP=dp+(ts*NumIndicies(idproc, nproc, maxIndex)*bfi[in].numQ);
						ComplexSingle
*d2tsP=d1tsP+NumIndicies(idproc, nproc, maxIndex);
						ComplexSingle tC;
```

```
                                    float tr;
                                    tC.a=cos((Pi/32.0)*ts);
                                    tC.b=sin((-Pi/32.0)*ts);
                                    while (x--) {
                                          float
snx=sin((x+StartIndex(idproc, nproc, maxIndex))*(Pi/bfi[in].sizeX));
                                          float
sn2x=sin(2*(x+StartIndex(idproc, nproc,
maxIndex))*(Pi/bfi[in].sizeX));
                                          CScalar(d1tsP[x],tC,snx);
                                          CMult(d2tsP[x],sn2x*tC,tC);
                                          }
                                    }
                              }
#endif

                              for(x2=0; x2<bfi[in].sizeX; x2++) {
                                    printf(" x2= %d/%d\n", x2,
bfi[in].sizeX);

                                    //Need to distribute Psi(x2)'s
                                    {long root; //who has the data
                                    for(root=0; (root<nproc)
                                          &&((x2<StartIndex(root,
nproc, maxIndex))
                                          ||(x2>=StartIndex(root+1,
nproc, maxIndex))); root++) ;

                                    if (idproc==root) {
                                          long ts=bfi[in].numSteps;

                                          while (ts--) {

p1x2P[ts]=dp[(ts*bfi[in].numQ)

*NumIndicies(idproc, nproc, maxIndex)
                                                      -
StartIndex(idproc, nproc, maxIndex)+x2];

p2x2P[ts]=dp[(ts*bfi[in].numQ+1)

*NumIndicies(idproc, nproc, maxIndex)
                                                      -
StartIndex(idproc, nproc, maxIndex)+x2];
                                                }

                                          }
                                    else {//nothing to be done

                                          }
                                    printf("  #%d Bcasting Psi data at
x2=%d… \n", root, x2);

                                    MPI_Bcast(p1x2P,
```

```
bfi[in].numSteps*2*2, MPI_FLOAT,
                                        root, MPI_COMM_WORLD);
                                  }

                                  printf("  Forming Psi12(x1, %d,
t)…\n", x2);
                                  //form a line of psi12
                                  {
                                  long ts=bfi[in].numSteps;
                                  while (ts--) {
                                        ComplexSingle
*p1tsP=&dp[(ts*bfi[in].numQ)

*NumIndicies(idproc, nproc, maxIndex)];
                                        ComplexSingle
*p2tsP=&dp[(ts*bfi[in].numQ+1)

*NumIndicies(idproc, nproc, maxIndex)];
                                        Complex *p12tsP=&p12P[ts
                                              *NumIndicies(idproc,
nproc, maxIndex)];
                                        long x1=NumIndicies(idproc,
nproc, maxIndex);
                                        while (x1--) {
                                              /*p1tsP[x1]*p2x2P[ts]
                                                    -
p2tsP[x1]*p1x2P[ts]; */
                                              myReal
a=p1tsP[x1].a*(myReal)p2x2P[ts].a-
(myReal)p1tsP[x1].b*(myReal)p2x2P[ts].b
                                                    -
(myReal)p2tsP[x1].a*(myReal)p1x2P[ts].a+(myReal)p2tsP[x1].b*(myReal)p1
x2P[ts].b;
                                              myReal
b=(myReal)p1tsP[x1].a*(myReal)p2x2P[ts].b+(myReal)p1tsP[x1].b*(myReal)
p2x2P[ts].a
                                                    -
(myReal)p2tsP[x1].a*(myReal)p1x2P[ts].b-
(myReal)p2tsP[x1].b*(myReal)p1x2P[ts].a;
                                              p12tsP[x1].a=a;
                                              p12tsP[x1].b=b;
                                              }
                                        }
                                  }
                                  {//Clearing second half of data
                                  long ts=bfi[in].numSteps;
                                  while (ts--) {
                                        Complex *p12tsP=

&p12P[(ts+bfi[in].numSteps)
                                              *NumIndicies(idproc,
nproc, maxIndex)];
                                        long x1=NumIndicies(idproc,
nproc, maxIndex);
```

```
                                        while (x1--) {
                                              p12tsP[x1].a=0;
                                              p12tsP[x1].b=0;
                                              }
                                        }
                                  }

                                  printf("  doing FFT %d times on a
%d array…\n",
                                        NumIndicies(idproc, nproc,
maxIndex),
                                        2L<<lgNumSteps);
/*                                DoFFTYCSingleN(p12P, lgNumSteps+1,
                                        NumIndicies(idproc, nproc,
maxIndex)); */
                                  DoFFTYCN(p12P, lgNumSteps+1,
                                        NumIndicies(idproc, nproc,
maxIndex));

                                  if (tempEigenFile) {
                                        //eigenfunction save
                                        long numWs=bfi[in].numSteps;
                                        if
(maxIndex*(myReal)maxIndex*(myReal)sizeof(Complex)*(myReal)numWs>(MaxE
igenMegs<<20)
                                              /*final file size
shouldn't exceed MaxEigenMegs*/)

numWs=(MaxEigenMegs<<20)/(maxIndex*(myReal)maxIndex*sizeof(Complex));
                                        printf("   saving data for
%d fermion eigenfunctions into temporary file…\n"
                                              , numWs);

                                  /*    {long w=8;
                                        while (w--)
                                        {long x1=NumIndicies(idproc,
nproc, maxIndex);
                                        while (x1--)
                                              p12P[x1+w*maxIndex].a=
                                                    0.01*exp(-
0.005*((x1-w)*(x1-w)+x2*x2));
                                        }
                                        }/**/

                                        fwrite(p12P, 1L,
                                              NumIndicies(idproc,
nproc, maxIndex)*sizeof(Complex)*numWs,
                                              tempEigenFile);
                                        if (ferror(tempEigenFile))
perror("fwrite etemp ");
                                        }

                                  printf("  summing into
output…\n");
```

```
                                    //sum into correlation output
                                    {long w=bfi[in].numSteps;
                                    while (w--) {
                                          Complex
*p12negwP=&p12P[(bfi[in].numSteps*2-1-w)
                                                *NumIndicies(idproc,
nproc, maxIndex)];
                                          Complex *p12wP=&p12P[w
                                                *NumIndicies(idproc,
nproc, maxIndex)];
                                          long x1=NumIndicies(idproc,
nproc, maxIndex);
                                          myReal sumA=0, sumB=0;
                                          while (x1--) {
                                          /*    sumA+=p12tsP[x1].a;
                                                sumB+=p12tsP[x1].b; */

sumA+=p12wP[x1].a*p12wP[x1].a+p12wP[x1].b*p12wP[x1].b;

sumB+=p12negwP[x1].a*p12negwP[x1].a+p12negwP[x1].b*p12negwP[x1].b;
                                                }
                                          cxOutP[cxOutRow*w].a+=sumA;
                                          cxOutP[cxOutRow*w].b+=sumB;
                                          }
                                    }

                                {
                                long etr=((bfi[in].sizeX-
(1+x2))*(MPI_Wtime()-startTime)/(1+x2));
                                long s, m, h;
                                char etrStr[64];
                                s=etr%60; etr/=60;
                                m=etr%60; etr/=60;
                                h=etr%24; etr/=24;
                                sprintf(etrStr, "ETR: %dd%2dh%2dm%2ds",
etr, h, m, s);
#ifdef __MEMORY__               //detects if on the Mac
                                logname(etrStr);
#endif
                                printf(" %s\n", etrStr);
                                }


                                }

                                printf(" summing output across
processors…\n");
                                MPI_Allreduce(cxOutP, p12P,
cxOutRow*bfi[in].numSteps*2,
                                      MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

                                {long i=bfi[in].numSteps*cxOutRow;
//copy output
```

```
                              while (i--)
cxOutP[i]=((Complex*)p12P)[i];
                              }

                              {
                              long etr=(MPI_Wtime()-startTime);
                              long s, m, h;
                              s=etr%60; etr/=60;
                              m=etr%60; etr/=60;
                              h=etr%24; etr/=24;
                              printf("Time Elapsed:
%dd%2dh%2dm%2ds\n", etr, h, m, s);
                              }

                              }
                              break;
                        case sizeof(double):
                              break;
                        default:
                              break;
                        }

                  break;
            case maxwellian:
            default:
                  /*
                                    printf("Doing FFTY… ");
                                    DoFFTYBabyQ(in, lgNumSteps,
idproc, nproc);
                                    printf("Done\n");

                                    {
                                    long w;

                                    for(w=0; w<bfi[in].numSteps; w++)
{
                                          if (!(w&0x3f))
printf("%d/%d\n", w, bfi[in].numSteps);
      /*                                  CorrelElemBabyQ(in, w, w,
cxOutP+(w*bfi[in].numQ),
                                                spinType, cxP, idproc,
nproc); * /
                                          }

                                    }/**/
                              {
                              ComplexSingle *dp=(ComplexSingle
*)bfi[in].dataP;
                              ComplexSingle
*scratchP=dp+bfi[in].numSteps*bfi[in].numQ
                                    *NumIndicies(idproc, nproc,
maxIndex);

                              //Clear second half
```

```
                              {long i=bfi[in].numSteps*bfi[in].numQ
                                    *NumIndicies(idproc, nproc,
maxIndex);
                              while (i--)
scratchP[i].b=scratchP[i].a=0;
                              }

#ifdef testdata
                              printf("Generating test data\n");
                              {long ts=bfi[in].numSteps;
                              while (ts--) {
                                    long i=bfi[in].numQ
                                          *NumIndicies(idproc, nproc,
maxIndex);
                                    ComplexSingle *dtsP=dp+(ts*i);
                                    ComplexSingle tC;
                                    tC.a=cos((Pi/32.0)*ts);
                                    tC.b=sin((-Pi/32.0)*ts);
                                    while (i--)
                                          dtsP[i]=tC;
                                    }
                              }
#endif
                              printf(" doing FFT on %d array %d
times\n",
                                    2L<<lgNumSteps,
NumIndicies(idproc, nproc, maxIndex)*bfi[in].numQ);
                              DoFFTYCSingleN(dp, lgNumSteps+1,
                                    NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ);

                              if (tempEigenFile) {
                                    //eigenfunction save
                                    long numWs=bfi[in].numSteps;
                                    if
(maxIndex*bfi[in].numQ*sizeof(ComplexSingle)*numWs>(MaxEigenMegs<<20)/
*32 megs*/)

numWs=(MaxEigenMegs<<20)/(maxIndex*bfi[in].numQ*sizeof(ComplexSingle))
;
                                    fwrite(dp, 1,
                                          NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ*sizeof(ComplexSingle)*numWs,
                                          tempEigenFile);
                                    }

                              printf("  summing into correlation
output…\n");
                              {long w=bfi[in].numSteps;
                              while (w--) {
                                    long q=bfi[in].numQ;
                                    while (q--) {
                                          ComplexSingle
*ptsqP=&dp[(w*bfi[in].numQ+q)
```

```
                                                *NumIndicies(idproc,
nproc, maxIndex)];
                                          long x1=NumIndicies(idproc,
nproc, maxIndex);
                                          float sumA=0, sumB=0;
                                          while (x1--) {
                                                /*sumA+=ptsqP[x1].a;
                                                sumB+=ptsqP[x1].b; */

sumA+=ptsqP[x1].a*ptsqP[x1].a+ptsqP[x1].b*ptsqP[x1].b;
                                                }

cxOutP[cxOutRow*w+q].a+=sumA*bfi[in].sizeX;

cxOutP[cxOutRow*w+q].b+=sumB*bfi[in].sizeX;
                                          }
                                    }
                              }

                              printf(" summing output across
processors…\n");
                              MPI_Allreduce(cxOutP, scratchP,
cxOutRow*bfi[in].numSteps*2,
                                    MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

                              {long i=bfi[in].numSteps*cxOutRow;
//copy output
                              while (i--)
cxOutP[i]=((Complex*)scratchP)[i];
                              }

                              }
            break;
            }
            }
      }

void GatherEigenBabyQData(short in, long lgNumSteps, long spinType,
const char *outEigenFN, FILE *tempEigenFile, long idproc, long nproc);
void GatherEigenBabyQData(short in, long lgNumSteps, long spinType,
const char *outEigenFN, FILE *tempEigenFile, long idproc, long nproc)
      {//Shuffles data from the temp files into the final eigenstate
output files
      //This rourine is primarily network and disk i/o intensive.  Not
much computation.
      long maxIndex=bfi[in].dataFrameSize/(2*bfi[in].floatSize);
      if (bfi[in].numQ<2) spinType=maxwellian;
      if (tempEigenFile) if (outEigenFN) {

            rewind(tempEigenFile); //Reset to beginning

      switch (spinType) {
            case fermion:
```

```
                  switch (bfi[in].floatSize) {
                        case sizeof(float):
                              {
                              Complex *dp=(Complex *)bfi[in].dataP;
                              Complex
*scratchP=dp+bfi[in].numSteps*bfi[in].numQ
                                          *NumIndicies(idproc, nproc,
maxIndex)*sizeof(ComplexSingle)/sizeof(Complex);
                              MPI_Request *mpiReqP=(MPI_Request
*)(scratchP+bfi[in].numSteps*bfi[in].numQ
                                          *NumIndicies(idproc, nproc,
maxIndex)*sizeof(ComplexSingle)/sizeof(Complex));
                              MPI_Status *mpiStatusP=(MPI_Status
*)(mpiReqP+nproc);
                              long lgElemSize=0,
                                    numWs=bfi[in].numSteps,
                                    chunkWs;

                              //Assuming sizeX is evenly divisible by
nproc
                              for(;
sizeof(Complex)*maxIndex/nproc>(4L<<lgElemSize); lgElemSize++) ;

                              if
(maxIndex*(myReal)maxIndex*(myReal)sizeof(Complex)*(myReal)numWs>(MaxE
igenMegs<<20)/*MaxEigenMegs megs*/)

numWs=(MaxEigenMegs<<20)/(maxIndex*(myReal)maxIndex*(myReal)sizeof(Com
plex));


chunkWs=bfi[in].numSteps*bfi[in].numQ*sizeof(ComplexSingle)/
                                          (maxIndex*sizeof(Complex)*nproc);

                              if (chunkWs<1) chunkWs=1;

                              if (chunkWs>numWs) chunkWs=numWs;

                              if (idproc) {
      long w;
      for(w=0; w<numWs; w+=chunkWs) {
            long sendWs=chunkWs, x2;
            if (w+sendWs>numWs) sendWs=numWs-w;

            printf("Sending data for eigenstates %d to %d out of
%d.\n", w, sendWs+w, numWs);

            for(x2=0; x2<maxIndex; x2++) {
                  fseek(tempEigenFile,
                        (w+x2*numWs)*NumIndicies(idproc, nproc,
maxIndex)*sizeof(Complex),
                        SEEK_SET);
                  fread(dp+x2*sendWs*NumIndicies(idproc, nproc,
maxIndex),
```

```
                        1L,
                        sendWs*NumIndicies(idproc, nproc,
maxIndex)*sizeof(Complex),
                        tempEigenFile);
                  }

            //Copy & transpose
            TransposeInMNOutData((long*)dp,
                  (long*)scratchP, lgElemSize, sendWs, maxIndex);

            MPI_Send(scratchP,
                  NumIndicies(idproc, nproc,
maxIndex)*maxIndex*sizeof(Complex)*sendWs,
                  MPI_BYTE, 0, idproc, MPI_COMM_WORLD);

            }

                                    }
                              else {//node 0

      long w;
      FILE *outEigenFP=fopen(outEigenFN, "wb");

      //write header
      if (outEigenFP) {
            BabyQBinaryHeaderStruct header;
            long i=sizeof(BabyQBinaryHeaderStruct)>>2,
*tp=(long*)&header;
            while (i--) tp[i]=0;

            rewind(outEigenFP);

            header.negVersion=-1L;
            header.dataOffset=sizeof(BabyQBinaryHeaderStruct);  //in
bytes from top of struct
            header.floatSize=sizeof(myReal);
            header.numSteps=numWs;
            header.numQ=maxIndex;
            header.sizeX=maxIndex;

            fwrite(&header, 1L,
                  sizeof(BabyQBinaryHeaderStruct),
                  outEigenFP);
            }

      for(w=0; w<numWs; w+=chunkWs) {
            long sendWs=chunkWs;
            if (w+sendWs>numWs) sendWs=numWs-w;

            printf("Receiving and processing data for eigenstates %d
to %d out of %d.\n", w, sendWs+w, numWs);

            {long x2;
            for(x2=0; x2<maxIndex; x2++) {
```

```
                int err;
                err=fseek(tempEigenFile,
                      (w+x2*numWs)*NumIndicies(idproc, nproc,
maxIndex)*sizeof(Complex),
                      SEEK_SET);
                if (err) printf("fseek err= %d\n", err);
                err=fread(dp+x2*sendWs*NumIndicies(idproc, nproc,
maxIndex),
                      1L,
                      sendWs*NumIndicies(idproc, nproc,
maxIndex)*sizeof(Complex),
                      tempEigenFile);
                if (ferror(tempEigenFile)) printf("fread err= %d
\n", ferror(tempEigenFile));
                }
          }

          {long i;
          for(i=1; i<nproc; i++) {

                MPI_Irecv(scratchP+StartIndex(i, nproc,
maxIndex)*maxIndex*sendWs,
                      NumIndicies(i, nproc,
maxIndex)*maxIndex*sizeof(Complex)*sendWs,
                      MPI_BYTE, i, i, MPI_COMM_WORLD,
                      mpiReqP+i);

                }
          }

     /*   {
          long x1;
          x1=maxIndex;
          while (x1--) {
                long tr1=(x1-w*2-1);
                long x2=maxIndex;
                tr1*=tr1;
                while (x2--) {
                      myReal tr=(x2-w-1);
                      dp[x2+x1*maxIndex*sendWs].a=0.000001*exp(-
0.001*(tr1+tr*tr));
                      dp[x2+x1*maxIndex*sendWs].b=0;
                      }
                }
          }*/

          //Copy & transpose
          TransposeInMNOutData((long*)dp, (long*)scratchP,
lgElemSize, sendWs, maxIndex);

          MPI_Waitall(nproc-1, mpiReqP+1, mpiStatusP+1);

          TransposeInMNOutData((long*)scratchP, (long*)dp,
lgElemSize, maxIndex*sendWs, nproc);
```

```
            if (outEigenFP) {
                  fwrite(dp, 1,
                        maxIndex*maxIndex*sizeof(Complex)*sendWs,
                        outEigenFP);
                  }

            }

      if (outEigenFP)
            fclose(outEigenFP);

                                    }

                              }
                              break;
                        case sizeof(double):
                              break;
                        default:
                              break;
                        }

                  break;
            case maxwellian:
            default:
                              {
                              ComplexSingle *dp=(ComplexSingle
*)bfi[in].dataP;
                              ComplexSingle
*scratchP=dp+bfi[in].numSteps*bfi[in].numQ
                                          *NumIndicies(idproc, nproc,
maxIndex);
                              MPI_Request *mpiReqP=(MPI_Request
*)(scratchP+bfi[in].numSteps*bfi[in].numQ
                                          *NumIndicies(idproc, nproc,
maxIndex));
                              MPI_Status *mpiStatusP=(MPI_Status
*)(mpiReqP+nproc);
                              long lgElemSize=0,
numWs=bfi[in].numSteps, chunkWs=bfi[in].numSteps/nproc;

                              //Assuming sizeX is evenly divisible by
nproc
                              for(;
bfi[in].dataFrameSize/nproc>(4L<<lgElemSize); lgElemSize++) ;

                              if
(maxIndex*bfi[in].numQ*sizeof(ComplexSingle)*numWs>(MaxEigenMegs<<20))

numWs=(MaxEigenMegs<<20)/(maxIndex*bfi[in].numQ*sizeof(ComplexSingle))
;

                              if (chunkWs>numWs) chunkWs=numWs;
```

```
                              if (idproc) {
      long w;
      for(w=0; w<numWs; w+=chunkWs) {
            long sendWs=chunkWs;
            if (w+sendWs>numWs) sendWs=numWs-w;

            MPI_Send(dp+NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ*w,
                  NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ*sizeof(ComplexSingle)*sendWs,
                  MPI_BYTE, 0, idproc, MPI_COMM_WORLD);

            }

                                    }
                              else {//node 0

      long w;
      FILE *outEigenFP=fopen(outEigenFN, "w");

      //write header
      if (outEigenFP) {
            BabyQBinaryHeaderStruct header;
            long i=sizeof(BabyQBinaryHeaderStruct)>>2,
*tp=(long*)&header;
            while (i--) tp[i]=0;

            header.negVersion=-1L;
            header.dataOffset=sizeof(BabyQBinaryHeaderStruct);  //in
bytes from top of struct
            header.floatSize=sizeof(float);
            header.numSteps=numWs;
            header.numQ=bfi[in].numQ;
            header.sizeX=maxIndex;

            fwrite(&header, 1,
                  sizeof(BabyQBinaryHeaderStruct),
                  outEigenFP);
            }

      for(w=0; w<numWs; w+=chunkWs) {
            long sendWs=chunkWs;
            if (w+sendWs>numWs) sendWs=numWs-w;

            {long i;
            for(i=1; i<nproc; i++) {

                  MPI_Irecv(scratchP+StartIndex(i, nproc,
maxIndex)*bfi[in].numQ*w,
                        NumIndicies(i, nproc,
maxIndex)*bfi[in].numQ*sizeof(ComplexSingle)*sendWs,
                        MPI_BYTE, i, i, MPI_COMM_WORLD,
                        mpiReqP+i);
```

```
                  }
            }

            //Copy & transpose
            TransposeInMNOutData((long*)dp+NumIndicies(idproc, nproc,
maxIndex)*bfi[in].numQ*w,
                  (long*)scratchP, lgElemSize, 1,
bfi[in].numQ*sendWs);

            MPI_Waitall(nproc-1, mpiReqP+1, mpiStatusP+1);

            TransposeInMNOutData((long*)dp, (long*)scratchP,
lgElemSize, bfi[in].numQ*sendWs, nproc);

            if (outEigenFP) {
                  fwrite(scratchP, 1,

maxIndex*bfi[in].numQ*sizeof(ComplexSingle)*sendWs,
                        outEigenFP);
                  }

            }

      if (outEigenFP)
            fclose(outEigenFP);

                                    }

                               }
            break;
            }
            }
      }

void CorrelateBabyqBinFileSpin(const char *inFileName, const char
*outFileName, const char *outEigenFileName, short spinType,
      long idproc, long nproc)
      {
      OSErr err;
      long nextCheckEscape=0;

      if (inFileName) {
            short fref;

            printf("Opening %s… \n", inFileName);
            fref=OpenBabyQBinFile(inFileName, idproc, nproc);

            if (fref) {

                  if (outFileName) {
                        Handle correlOutH=nil;
                        long numQ=GetNumQ(fref),
numSteps=GetNumSteps(fref);
                        unsigned long lgNumQ, lgNumSteps;
```

```
                        switch (spinType) {
                              case fermion:
                              case boson:
                                    if (numQ>1) numQ=1;
                                    break;
                              case maxwellian:
                              default: break;
                              }

                        for(lgNumQ=0; numQ>(1L<<lgNumQ); lgNumQ++) ;
                        for(lgNumSteps=30; numSteps<(1L<<lgNumSteps);
lgNumSteps--) ;

                        correlOutH=NewHandle(sizeof(Complex)*
                              (numSteps+2)<<lgNumQ);

                        if (correlOutH) {
                              Complex *correlOutTop;
                              Complex *cxP;
                              long leaving=0;
                              FILE *tempEigenFile=nil;
                              char tEFN[FILENAME_MAX]="";

                              MoveHHi(correlOutH);
                              HLock(correlOutH);

                              {long *tp=(long*)*correlOutH,
i=GetHandleSize(correlOutH)>>2;
                                    while(i--) tp[i]=0; }
                              correlOutTop=(Complex*)*correlOutH;
                              cxP=(correlOutTop+(numSteps<<lgNumQ));

                              printf("Attempting to load all data…");
                              if (LoadAllBabyQData(fref, idproc,
nproc)) {

                                    printf("Renormalizing data…");
                                    RenormalizeBabyQData(fref,
spinType, idproc, nproc);

                                    if (outEigenFileName) {
                                          sprintf(tEFN, "%sTemp%d/%d",
outEigenFileName, idproc, nproc);
                                          tempEigenFile=fopen(tEFN,
"w+b");
                                          if (tempEigenFile) {
                                                rewind(tempEigenFile);
                                                }
                                          }

                                    printf("Doing Correlation of %d
time steps…\n", numSteps);
```

```
                                    CorrBabyQSpin(fref, lgNumSteps,
spinType,
                                          correlOutTop, 1L<<lgNumQ,
tempEigenFile,
                                          idproc, nproc);

                                    }
                              else {
                                    leaving=1;
                                    }

                  if (!leaving) if (!idproc) {  FILE *outfp=nil;
                  //Only node 0 writes

/*                if (tSFP.sfReplacing)
                        FSpDelete(&tSFP.sfFile);
                  FSpCreate(&tSFP.sfFile, 'R*ch', 'TEXT',
tSFP.sfScript); */
                  printf("Opening out file…\n");
            //    FSpOpenDF(&tSFP.sfFile, fsWrPerm, &outref);
                  outfp=fopen(outFileName, "w");

                  printf("Writing output…\n");

#ifdef mathematica
                  {long q;
                  for(q=0; q<numQ; q++) {
                        if (numQ>1) {
                           fprintf(outfp, "%s", q?",\n":"{"/*}*/);
                           }
                                    {
                                    long tau;

                                    for(tau=0; tau<(numSteps); tau++)
{
                                          myReal
adjust=1.0/*((myReal)numSteps-tau)*/;
                                          myReal
ca=correlOutTop[q+(tau<<lgNumQ)].a*=adjust,

cb=correlOutTop[q+(tau<<lgNumQ)].b*=adjust;
                                          long
ex=ca?log10(fabs(ca)):0;

                                          fprintf(outfp, "%s",
tau?",\n":"{"/*}*/);

                                          ca*=exp(-log(10.0)*ex);
                                          fprintf(outfp, "%-.12g *
10^%d", ca, ex);
                                          ex=cb?log10(fabs(cb)):0;
                                          fprintf(outfp, " + I* ");
                                          cb*=exp(-log(10.0)*ex);
                                          fprintf(outfp, "%-.12g *
```

```
10^%d", cb, ex);

                                        }
                                  }

                            fprintf(outfp,  /*{*/"}");


                      }
                if (numQ>1) {
                      fprintf(outfp, /*{*/"}");
                      }
                }
#else
                {long tau;
                for(tau=0; tau<(numSteps); tau++) {
                                  {
                                  long q;

                                  for(q=0; q<numQ; q++) {
                                        myReal
adjust=1.0/*((myReal)numSteps-tau)*/;
                                        myReal
ca=correlOutTop[q+(tau<<lgNumQ)].a*=adjust,

cb=correlOutTop[q+(tau<<lgNumQ)].b*=adjust;

                                        fprintf(outfp, "%-16.12g",
ca);
                                        fprintf(outfp, "\t");
                                        fprintf(outfp, "%-16.12g",
cb);
                                        fprintf(outfp, "\t");

                                        }
                                  }

                            fprintf(outfp,  "\n");

                      }
                }
#endif

            if (0)      {
                  //Now, Fourier transform the data
                  printf("Doing FFT…");
                  DoFFTYC(correlOutTop, lgNumSteps, lgNumQ);
                  printf("Writing FT…");
                  {long q;

                  fprintf(outfp, "\n\n");

                  for(q=0; q<numQ; q++) {
                        if (numQ>1) {
```

```
                              fprintf(outfp, "%s", q?",\n":"{" /*}*/);
                              }
                                    {
                                    long tau;

                                    for(tau=0; tau<(1<<(lgNumSteps-
1)); tau++) {

                                          fprintf(outfp, "%s",
tau?",\n":"{"/*}*/);

                                          fprintf(outfp, "%-16.12g +
I* %-16.12g",

correlOutTop[q+(tau<<lgNumQ)].a,

correlOutTop[q+(tau<<lgNumQ)].b);
                                          }
                                    }

                              fprintf(outfp, /*{*/"}");


                        }
                  if (numQ>1) {
                        fprintf(outfp, /*{*/"}");
                        }
                  }
                  }

#undef fprint


                  printf("\nClosing out file\n");
                  fclose(outfp);
                              }

                              if (tempEigenFile) {
                                    printf("\nGathering Eigenstate
data to node zero…\n");
                                    GatherEigenBabyQData(fref,
lgNumSteps, spinType,
                                          outEigenFileName,
tempEigenFile, idproc, nproc);

                                    fclose(tempEigenFile);
                                    remove(tEFN);
                                    }

                              HUnlock(correlOutH);
                              DisposeHandle(correlOutH);
                              correlOutH=nil;
                              }
                        else printf(" Out of Memory!\n");
```

```
                    }

                printf("Closing…\n");
                CloseBabyQBinFile(fref, idproc, nproc);
                }
        else {
                printf("Unable to open input babyq data file.\n");
                perror(inFileName);
                }

        }

  }
```

Listing G. Quantum data reader, correlation analysis, and eigenstate extraction code.

XI. References

1. M. Planck, “Ueber irreversible Strahlungsvorgänge”, *Ann. d. Phys.*, **1**, p. 69, (1900); M. Planck, “Entropie und Temperatur strahlender Wärme”, *Ann. d. Phys.*, **1**, p. 719, (1900).
2. A. Einstein, “Über einen die Erzeugung und Verwandlung des Lichtes betreffenden heuristischen Gesichtspunkt”, *Ann. d. Phys.*, **17**, p. 132, (1905).
3. L. de Broglie, “Recherches sur la théorie des Quanta”, *Ann. d. Phys.*, (10) **3**, p. 22, 1925 (Thèses, Paris 1924).
4. E. Schrödinger, “Quantiseirung als Eigenwertproblem”, *Ann. d. Phys.*, **79**, p. 361, (1926); E. Schrödinger, *Ann. d. Phys.*, **79**, p. 489, (1926); E. Schrödinger, *Ann. d. Phys.*, **80**, p. 437, (1926); E. Schrödinger, *Ann. d. Phys.*, **81**, p. 109, (1926).
5. Commonly known today as “WKB”, the method, as it was originally

presented, was designed specifically for solving Schrödinger's equation of wave mechanics. It is in such common use today that citations to the original references are difficult to find. The original three papers are:

- G. Wentzel, "Eine Verallgemeinerung der Quantenbedingungen für Zwecke der Wellenmechanik", *Zeitschrift für Physik*, **38**, p. 518, (1926);
- H. A. Kramers, "Wellenmechanik und halbzahlige Quantisierung", *Zeitschrift für Physik*, **39**, p. 828, (1926);
- L. Brillouin, "La mècanique ondulatoire de Schrödinger; une mèthode gènèrale de rèsolution par approximations successives", *Comptes Rendus (Académie de Sciences)*, **183**, p. 24, (1926).

6. J. H. Van Vleck, "The Correspondence Principle in the Statistical Interpretation of Quantum Mechanics", *Proc. Natl. Acad. Sci. (USA)*, **14**, p. 178 (1928).
7. P. M. Dirac, *Physikalische Zeitschrift der Sowjetunion*, **3**, p. 1 (1933).
8. R. P. Feynman, *Rev. Mod. Phys.*, **20**, p. 367 (1948).
9. R. P. Feynman and A. R. Hibbs, *Quantum Mechanics and Path Integrals*, (McGraw-Hill, Inc., New York, 1965).
10. M. C. Gutzwiller, "Phase-Integral Approximation in Momentum Space and the Bound States of an Atom", *J. Math. Phys.*, **8**, p. 1979 (1967).
11. E. Heller, *J. Chem. Phys.*, **94**, p. 2723, (1991).
12. S. Tomsovic and E. Heller, *Phys. Rev. Lett.*, **67**, 6, p. 664, (1991).

13. E. Heller and S. Tomsovic, "Postmodern Quantum Mechanics", *Physics Today*, **46**, No. 7, p. 38, (1993), and references therein.

14. S. Tomsovic and E. Heller, *Phys. Rev. Lett.*, **70**, p. 1405, (1993).

15. S. Tomsovic and E. J. Heller, "Long-time semiclassical dynamics of chaos: The stadium billiard", *Phys. Rev. E*, **47**, p. 282, (1993).

16. V. P. Maslov and M. V. Fedoriuk, *Semiclassical Approximations in Quantum Mechanics*, (Reidel, Dordrecht, 1981), English translation.

17. L. S. Schulman, *Techniques and Applications of Path Integration* (Wiley, New York, 1981) and references therein.

18. M. A. Sepúlveda, S. Tomsovic, E. Heller, *Phys. Rev. Lett.*, **69**, p. 402, (1992).

19. M. C. Gutzwiller, *Chaos in Classical and Quantum Mechanics*, (Springer-Verlag, New York, 1990).

20. O. Buneman, *Physical Review*, **115**, p. 503, (1959).

21. J. M. Dawson, Princeton University Plasma Physics Laboratory, *Project Matterhorn Rept.*, MATT-4, (1959).

22. J. M. Dawson, Princeton University Plasma Physics Laboratory, *Project Matterhorn Rept.*, MATT-31, (1960).

23. J. M. Dawson, *Physics of Fluids*, **5**, p. 445, (1962).

24. J. M. Dawson in *Methods in Computational Physics*, **9**, Ed. by Alder, Fernbach, and Rotenburg (Academic Press, New York, 1970), p. 1.

25. R. W. Hockney, *Physics of Fluids*, **9**, p. 1826, (1966).

26. J. W. Cooley and J. W. Tukey, *Mathematics of Computation*, **19**, 297, (1965).

27. J. M. Dawson, C. G. Hsi, and R. Shanny, Princeton University Plasma Physics Laboratory, *Project Matterhorn Rept.*, MATT-719, (1969).

28. J. M. Dawson, *Rev. of Mod. Phys.*, **55**, p. 403, (1983).

29. J. M. Dawson, V. K. Decyk, R.D. Sydora, and P. Liewer, "High-Performance Computing and Plasma Physics", *Physics Today*, **46**, No. 3, p. 64, (1993).

30. V. K. Decyk, "Parallel Processing of Particle Simulation Models", Proceedings of the International Workshop on Plasma Physics, Pichl, Austria, Feb. 1992, in *Current Topics in Astrophysical and Fusion Plasma Research*, M.F. Heyn and W. Kernbichler, ed. [dbu-[Dbu Graz, Austria, 1992], p. 187.

31. J. M. Dawson and V. K. Decyk, "Particle Modeling of Plasmas on Super-computers", *International Journal of Supercomputer Applications*, **1**, p. 24, (1987).

32. J. M. Dawson, "The Numerical Tokamak: A Grand Challenge for Fusion Plasma Modeling", *Proc. of IAEA Technical Committee Meeting on Advances in Simulation and Modeling Thermonuclear Plasmas*, Montreal, Quebec, Canada, 1 (1992).

33. J. M. Dawson, "Computer Modeling of Plasma: Past, Present, and Future", *Phys. Plasmas*, **2**, p. 6 (1995).

34. V. K. Decyk, “How to Write (Nearly) Portable Fortran Programs for Parallel Computers”, *Computers in Physics*, **7**, p. 418, (1993).

35. V. K. Decyk, “Skeleton PIC Codes for Parallel Computers”, *Computer Physics Communications*, **87**, p. 87, (1995).

36. V. K. Decyk, D. E. Dauger, P. R. Kokelaar, “How to Build an AppleSeed: A Parallel Macintosh Cluster for Numerically Intensive Computing”, *Physica Scripta*, **T84**, p. 85-88, (2000).

37. R. B. Gerber, V. Buch, and M. A. Ratner, “Simplified time-dependent self consistent field approximation for intramolecular dynamics”, *Chem. Phys. Lett.*, **91**, p. 173, (1982).

38. U. Peskin and M. Steinberg, “A temperature-dependent Schrödinger equation based on a time-dependent self consistent field approximation”, *J. Chem. Phys.*, **109**, p. 704, (1998).

39. K. M. Christoffel and P. Brumer, “Quantum and classical dynamics in the stadium billiard”, *Phys. Rev. A*, **33**, p. 1309, (1986).

40. E. J. Heller, “Frozen Gaussians: a very simple semiclassical approximation”, *J. Chem. Phys.*, **75**, p. 2923, (1981).

41. N. Makri, “Time-dependent self-consistent field approximation with explicit 2-body correlations”, *Chem. Phys. Lett.*, **169**, p. 541, (1990).

42. M. F. Herman, E. Kluk, and H. L. Davis, “Comparison of the propagation of semiclassical frozen Gaussian wave functions with

quantum propagation for a highly excited anharmonic oscillator," *J. Chem. Phys.*, **84**, p. 326, (1986).

43. M. S. Child, *Semiclassical mechanics with molecular applications*. Oxford: Clarendon, (1991).

44. K. G. Kay, "Semiclassical propagation for multidimensional systems by an initial value method", *J. Chem. Phys.*, **101** (3), p. 2250, (1994).

45. A. R. Walton and D. E. Manolopoulos, "A new semiclassical initial value method for Franck-Condon spectra", *Mol. Phys.*, **87**, p. 961, (1996).

46. X. Sum, H. B. Wang, and W. H. Miller, "Semiclassical theory of electronically nonadiabatic dynamics: Results of a linearized approximation to the initial value representation", *J. Chem. Phys.*, **109**, p. 7064, (1998).

47. K. Thompson and N. Makri, "Rigorous forward-backward semiclassical formation of many-body dynamics", *Phys. Rev. E*, **59**, p. 4729, (1999).

48. L. Kaplan and E. J. Heller, "Overcoming the Wall in Semiclassical Baker's Map", *Phys. Rev. Lett.*, **76**, p. 1453, (1996).

49. G. Campolieti and P. Brumer, "Semiclassical initial value approach for chaotic long-lived dynamics", *J. Chem. Phys.*, **109**, p. 2999, (1998).

50. N. T. Maitra and E. J. Heller, "Semiclassical amplitudes: Supercaustics and the whisker map", *Phys. Rev. A*, **61**, p. 0212107-1, (1999).

51. M. Brack and R. K. Bhaduri, *Semiclassical Physics* (Addison-Wesley, Inc.,

New York, 1997) and references therein.

52. E. J. Heller, in *Chaos and Quantum Physics*, Proceedings from Les Houchs 1989 (North-Holland, Amsterdam, 1989).

53. F. P. Simotti, E. Vergini, and M. Saraceno, "Quantitative study of scars in the boundary section of the stadium billiard", *Phys. Rev. E*, **56**, p. 3859, (1997).

54. J. S. Townsend, *A Modern Approach to Quantum Mechanics* (McGraw-Hill, Inc., New York, 1992).

55. P. C. Liewer and V. K. Decyk, "A General Concurrent Algorithm for Plasma Particle-in-Cell Codes", *J. Computational Phys.*, **85**, p. 302, (1993).

56. Parallel 1-D FFT routine written by Viktor K. Decyk, unpublished.

57. V. K. Decyk, S. R. Karmesin, A. de Boer, and P. C. Liewer, "Optimization of particle-in-cell codes on reduced instruction set computer processors", *Computers In Physics*, **10**, p. 290, (1996).

58. D. Neuhauser, *J. Chem. Phys.*, **93** (4), p. 2611, (1990).

59. V. K. Decyk, "Wave-particle diagnostics for plasma simulation", *Space Science Reviews*, **42**, p. 113, (1985).

60. See http://www.stat.ucla.edu/research/gSCAD/

61. See http://exodus.physics.ucla.edu/appleseed/

62. S. Ichimaru, *Basic Principles of Plasma Physics: A Statistical Approach* (W. A. Benjamin, Inc., Reading, Massachusetts, 1973).

63. J. M. Dawson, “Radiation from Plasmas”, *Adv. Plasma Phys.*, **1**, p. 1 (1968).

64. J. M. Dawson, “Irreversible Statistical Mechanics”, Course 215B, Non-equilibrium Statistical Mechanics, APS, UCLA (1987).

65. N. Rostoker, R. Aamodt, and O. Eldridge, *Ann. Phys.*, **31**, p. 243 (1965).

66. A. G. Sitenko and A. A. Gurin, *JETP*, **22**, p. 1089 (1966).

67. A. I. Akheizer, I. A. Akhiezer, and A. G. Sitenko, *JETP*, **14**, p. 462 (1961).

68. A. G. Sitenko, *Electromagnetic Fluctuations in Plasma* (Academic press, New York, 1967).

69. A. I. Akheizer, I. A. Akhiezer, R. V. Plovin, A. G. Sitenko, and K. N. Stepanov, *Plasma Electrodynamics* (Pergamon Press, Oxford, 1975), Vol. 2.

70. M. Opher and R. Opher, *Phys. Rev. Lett.*, **79**, p. 2628, (1997).

71. M. Opher and R. Opher, *Phys. Rev. Lett.*, **82**, p. 4835, (1999).

72. M. Opher and R. Opher, *Astrophys. J.*, **535**, p. 473, (2000).

73. M. Opher and R. Opher, astro-ph / 00063262. (submitted to Phys. Rev. Lett.)

74. M. Opher, L. O. Silva, D. E. Dauger, V. K. Decyk and J. M. Dawson, submitted to Physics of Plasmas, October 2000.

75. F. Haas, G. Manfredi, and M. Feix, “Mutlistream model for quantum plasmas”, *Phys. Rev. E*, **62**, p. 2763 (2000).

76. A. B. Langdon, “Kinetic theory for fluctuations and noise in computer simulation of plasma”, *Physics of Fluids*, **22**, No. 1, p. 163, (1979).

77. V. K. Decyk and J. E. Slottow, “Supercomputers in the classroom”, *Computers in Physics*, **3**, No. 2, p. 50, (1989).

78. J. M. Dawson and T. Nakayama, “Kinetic Structure of a Plasma”, *Physics of Fluids*, **9**, No. 2, p. 252, (1966).

79. Such as the Atom in a Box shareware available at http://dauger.com/orbitals/

80. A. Einstein, *Ann. d. Phys.*, “Zur Elektrodynamik bewegter Körper”, **17**, 861, (1905).

81. R. D. Ferraro, P. C. Liewer, and V. K. Decyk, “Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code”, *J. Computational Phys.*, **109**, p. 329, (1993).

82. P. C. Liewer, V. K. Decyk, J. M. Dawson, and B. Lembege, “Numerical Studies of Electron Dynamics in Oblique Quasi-Perpendicular Collisionless Shock Waves”, *J. Geophysical Research*, **96**, p. 9455, (1991).

83. D. E. Dauger, “Simulation and study of Fresnel diffraction for arbitrary two-dimensional apertures”, *Computers In Physics*, **10** (6), p. 591, (1996).

84. C. K. Chui, *Introduction to Wavelets* (Academic Press, New York, 1992).