Numerical solution of the time-dependent Schrödinger equation for a 2D electron waveguide

Term project for PHY 381C: Computational Physics

Markus A. Lischka

May 10, 1997

Abstract

Electrons at a GaAs/GaAlAs interface behave like a 2D electron gas. Confining these electrons to leads gives an electron waveguide in which, at very low temperatures, quantum interference effects are dominant and the motion of the electrons are governed by the single-electron Schrödinger equation. In the following, we will explain how to numerically simulate the Schrödinger equation using a finite-difference approximation and present some 1D and 2D results.

Contents

1 Introduction: conductance in an electron waveguide

A 2D electron waveguide typically consists of leads connecting the region of interest with electron reservoirs. The simplest example is shown in Fig. 1 where two leads are connected to a cavity. The shown boundaries are hard-wall boundaries thus confining the electrons inside the waveguide.

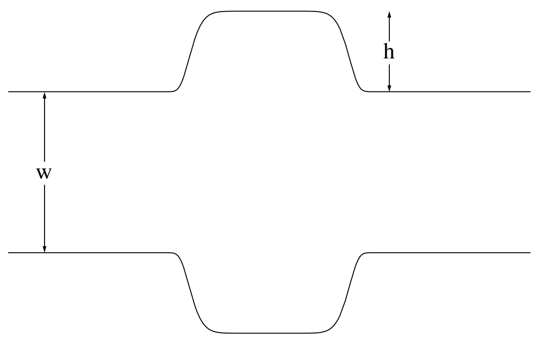

Figure 1: 2D electron waveguide

These waveguides with lengths in the nanometer scale can be constructed at a semiconductor interface (e.g. GaAs/GaAlAs) where a two-dimensional electron gas is formed. At very low temperatures, the electron motion is ballistic, and quantum interference effects can be observed. The only scattering left in this setup that influences the electron motion is scattering from the boundaries. In 1957, Landauer proposed the formula [11]

$$G = \frac{e^2}{h} T(E_F) \tag{1}$$

which relates the conductance of the waveguide to the transmission coefficient of an electron at the Fermi energy and which can be derived from microscopic linear response theory. Note that (1) relates a statistical quantity of a many-body system (the conductance) with a quantity of a single-electron quantum mechanical system (the transmission coefficient).

Therefore, it seems to be possible to build quantum devices exploiting the fact that the electron motion is determined by the device's boundaries. A quantum transistor based on this setup has been proposed e.g. by [18, 19]. Furthermore, the transmission coefficient in this 2D waveguide shows interesting resonance structures (cf. Fig. 2). The transmission function exhibits sudden dips to zero with varying widths. For the case of a rectangular cavity, this happens at the eigenenergies of the cavity with no leads connected.

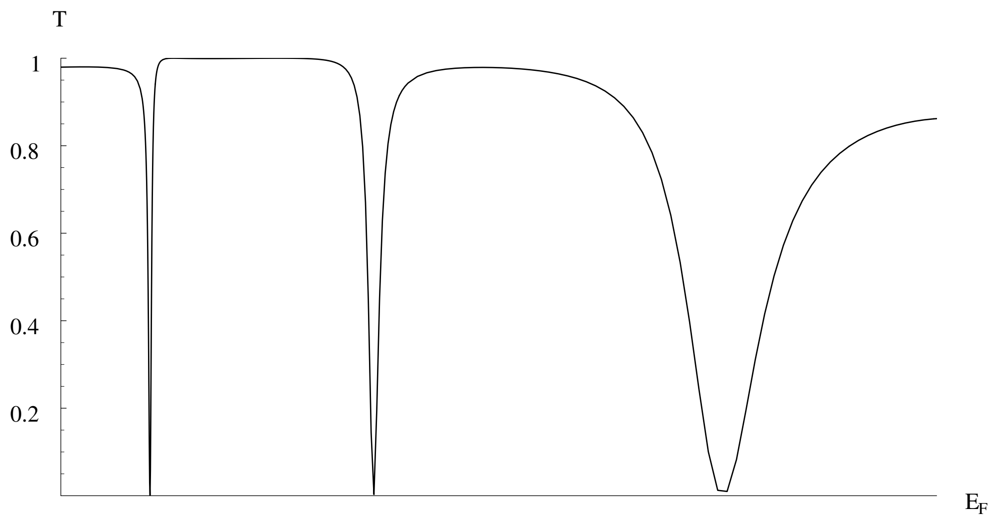

Figure 2: Transmission in a waveguide with a rectangular cavity [14]

In the following, we will describe how to simulate such a system dynamically based on a FDA approximation of the Schrödinger equation.

2 The Schrödinger equation with arbitrary metric

To include the appropriate boundary conditions of the waveguide in our simulation, we will perform a transformation of the physical system in cartesian coordinates with x extending from $-\infty$ to ∞ and y being confined as shown in Fig. 1 to a set of curvilinear coordinates with range $[0, 1] \times [0, 1]$. One possible mapping is

$$
\begin{aligned}
x &= \tan\left(\pi\left(u - \frac{1}{2}\right)\right) \\
y &= (w + h\xi(u))\, v
\end{aligned}
$$

where x, y are our cartesian coordinates and u, v are our new curvilinear coordinates. ξ can be chosen to describe the cavity, e.g.

$$\xi(x) = e^{-\frac{x^{2p}}{2\sigma^2}}$$

with appropriate parameters p, σ, w and h determining the geometry of the waveguide (cf. Fig. 1). The resulting metric is

$$g_{ij} = \begin{pmatrix} \pi^2 \sec\left(\pi\left(u-\frac{1}{2}\right)\right)^4 + h^2(2v-1)^2\xi'(u)^2 & 2h(2v-1)\left(w+h\xi(u)\right)\xi'(u) \\ 2h(2v-1)\left(w+h\xi(u)\right)\xi'(u) & 4\left(w+h\xi(u)\right)^2 \end{pmatrix}$$

The Schrödinger equation in curvilinear coordinates is given by

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m^*}g^{-1/2}\frac{\partial}{\partial u_i}\left(g^{1/2}g^{ij}\frac{\partial\psi}{\partial u_j}\right) + V\psi \tag{2}$$

where $g = \det g_{ij}$ [7], or equivalently

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m^*}\left(a_{20}\frac{\partial^2}{\partial u^2} + a_{02}\frac{\partial^2}{\partial v^2} + a_{11}\frac{\partial^2}{\partial u\partial v} + a_{10}\frac{\partial}{\partial u} + a_{01}\frac{\partial}{\partial v}\right)\psi + V\psi \tag{3}$$

where the coefficients a_{ij} are obtained by inserting g_{ij} in (2).

3 Computational implementation

3.1 Computational units

Before starting any numerical calculations, it is in general convenient to perform a variable transformation to unitless variables. In our case we have to write the Schrödinger equation of a one particle system,

$$i\hbar\frac{\partial\psi}{\partial t} = -\frac{\hbar^2}{2m^*}\nabla^2\psi + V\psi,$$

with dimensionless coordinates. As we are effectively dealing with a one particle system, we can use the mass of the particle as a scaling parameter. We will use the following transformation:

$$\begin{aligned} M: &\quad \frac{1}{m^*}m_{\text{phys}} \mapsto m_{\text{comp}} = 1 \\ L: &\quad \frac{m^*c}{\hbar}x_{\text{phys}} \mapsto x_{\text{comp}} \\ T: &\quad \frac{m^*c^2}{\hbar}t_{\text{phys}} \mapsto x_{\text{comp}} \end{aligned}$$

Especially, using $m^* = 0.05 m_e$ for the effective electron mass in GaAs, we get these relations between computational and physical units:

$$\begin{aligned} M: &\quad m_{phys} = 4.555 \times 10^{-32}\text{kg}\; m_{\text{comp}} \\ L: &\quad x_{phys} = 7.723 \times 10^{-12}\text{m}\; x_{\text{comp}} \\ T: &\quad t_{phys} = 2.576 \times 10^{-20}\text{s}\; t_{\text{comp}} \end{aligned}$$

In these new computational variables the Schrödinger equation reads

$$\Rightarrow \quad i\frac{\partial \psi}{\partial t_{\text{comp}}} = -\frac{1}{2}\nabla^2_{\text{comp}}\psi + V_{\text{comp}}\psi$$

where $V_{\text{comp}} = \frac{V_{\text{phys}}}{m^* c^2}$. Subsequently, we will drop the comp index and refer to

$$i\frac{\partial \psi}{\partial t_{\text{comp}}} = H\psi = -\frac{1}{2}\nabla^2_{\text{comp}}\psi + V_{\text{comp}}\psi \tag{4}$$

as the Schrödinger equation to be solved.

3.2 Crank-Nicholson scheme

The formal solution of eq. (4) is given by

$$\psi(x,t) = e^{-iHt}\psi(x,0)$$

where the exponential operator is defined in the usual way in terms of the Taylor's series as

$$e^{-iHt} = \sum_{k=0}^{\infty} \frac{(-it)^k}{k!} H^k.$$

For a finite, but small time step Δt we therefore get

$$e^{-iH\Delta t} = 1 - iH\Delta t + O(\Delta t^2)$$

One characteristic of the Schrödinger equation is that the norm of the wave function,

$$\|\psi\| = \int_{-\infty}^{\infty} |\psi|^2 \, dx,$$

does not change in time. It is therefore desirable that the finite difference approximation to (4) has the same property. For this purpose note

Theorem 1 (Cayley transform) *If A is a selfadjoint operator, then*

$$V = (I + iA)(I - iA)^{-1}$$

is a unitary operator. [9]

If we write our approximation to e^{-iHt} in this form, we are guaranteed the unitarity of our approximation:

$$\begin{aligned} & e^{\frac{1}{2}iHt}\psi(x,t) = e^{-\frac{1}{2}iHt}\psi(x,0) \\ \Rightarrow \quad & (1 + \frac{1}{2}iHt)\psi(x,t) \approx (1 - \frac{1}{2}iHt)\psi(x,0) \\ \Rightarrow \quad & e^{-iHt} \approx \frac{1 - \frac{1}{2}iHt}{1 + \frac{1}{2}iHt} \end{aligned}$$

Rewriting the last equation in the usual notation for finite-difference approximations (FDA) with the lower index j denoting the space grid points and the upper index n the discrete time points we get

$$(1 + iH\frac{\Delta t}{2})\psi_j^{n+1} = (1 - iH\frac{\Delta t}{2})\psi_j^n \tag{5}$$

which is just the implicit Crank-Nicholson scheme for solving diffusion problems [4]. We therefore already know that this method is stable and accurate in time to order $O(\Delta t^2)$. Furthermore, by the above derivation we know it is unitary, or in other words preserving the coefficients of any Fourier components of our wavefunction. The only disadvantage of this scheme is that the updated wavefunction at time step $n+1$ is given implicitly, i.e. we need to solve a linear system to update it.

3.3 Finite difference approximation for H

To have a full finite-difference approximation to (4) we still need to find a discretized version of our Hamiltonian operator H. We therefore have to approximate the derivatives appearing in H by finite difference schemes. As we are dealing with an arbitrary metric, all possible derivatives in both x- and y-direction need to be approximated. To have $O(h^2)$ accuracy in space we employ the following differencing schemes:

$$
\begin{aligned}
\frac{\partial^2 \psi_{ij}}{\partial x^2} &= \frac{1}{h_x^2}\left(\psi_{i-1,j} - 2\psi_{ij} + \psi_{i+1,j}\right) \\
\frac{\partial \psi_{ij}}{\partial x} &= \frac{1}{2h_x}\left(\psi_{i+1,j} - \psi_{i-1,j}\right) \\
\frac{\partial^2 \psi_{ij}}{\partial y^2} &= \frac{1}{h_y^2}\left(\psi_{i,j-1} - 2\psi_{ij} + \psi_{i,j+1}\right) \\
\frac{\partial \psi_{ij}}{\partial y} &= \frac{1}{2h_y}\left(\psi_{i,j+1} - \psi_{i,j-1}\right) \\
\frac{\partial^2 \psi_{ij}}{\partial x \partial y} &= \frac{1}{4h_x h_y}(\psi_{i+1,j+1} - \psi_{i+1,j-1} \\
&\quad -\psi_{i-1,j+1} + \psi_{i-1,j-1})
\end{aligned}
$$

where h_x and h_y denote the mesh spacing in x- and y-direction respectively and which can be found by Taylor expanding ψ at the involved grid points. Putting all these terms together and using the abbreviated notation of (3), we obtain the full stencil shown in Fig. 3.

3.4 Boundary conditions

We will simulate the Schrödinger equation on the unit square $[0,1]\times[0,1]$. On the borders of our computational domain we will impose hard wall boundary conditions. Hence the wavefunction has to vanish there:

$$
\begin{aligned}
\psi_{i0}^n = \psi_{iN_y}^n = 0 \qquad \forall i, n \\
\psi_{0j}^n = \psi_{N_x j}^n = 0 \qquad \forall j, n
\end{aligned}
$$

From one time step to the next, we therefore have to update the interior points according to our stencil, whereas on the boundaries the wavefunction just stays zero.

"Open" boundary conditions are then simulated by mapping the interval $[0,1]$ onto the real axis as described in section 2.

3.5 Solving the implicit FD approximation

In pseudo code the algorithm reads:

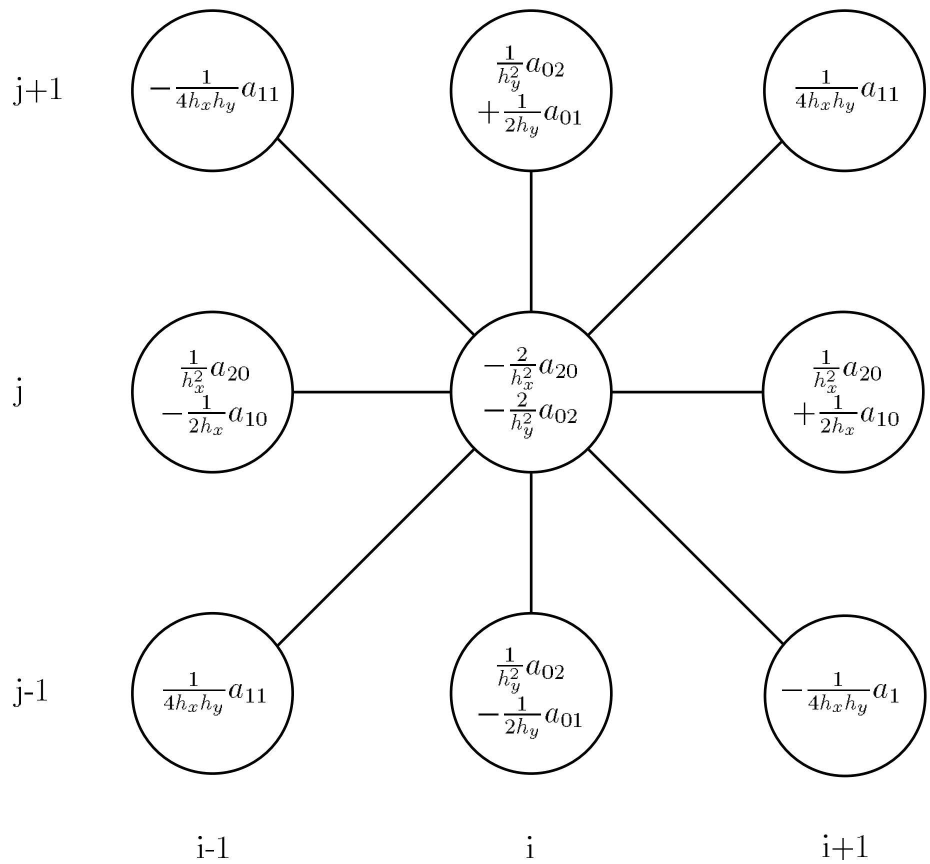

Figure 3: Stencil

```
Parse command-line input
Initialize wave function
Do requested number of timesteps:
    Apply Crank-Nicholson scheme:
        Calculate advanced ψ for Δt/2
        Calculate time propagator matrix for −Δt/2
        Solve linear equation
    Check norm
    Output solution
```

The program accepts the following command-line input:

```
wg <tfinal> <levelx> <levely> <olevel> <dt/dx>
   <k> <x0> <sigma0> <c1> [<c2> ...]
```

where

- <tfinal> is the final simulation time.
- <levelx> and <levely> are the discretization levels for x- and y-direction. The number of grid points in each direction is $n = 2^{\texttt{<level>}} + 1$.
- <olevel> is the output level. Output is produced every $2^{\texttt{<levelx>}-\texttt{<olevel>}}$ time steps.
- <dt/dx> is the Courant number specifying the ratio of the simulation time step Δt to the mesh spacing in x-direction $h_x = n_x^{-1}$.
- <k>, <x0> and <sigma0> specify the Gaussian wave packet in x-direction:

$$\psi_x = \exp\left(-\frac{(x - < \texttt{x0} >)^2}{2 < \texttt{sigma0} >^2}\right)$$

- <c1>, <c2>, ... describe the wave function in y-direction:

$$\psi_y = \sum_n < \texttt{cn} > \sin(n\pi y)$$

Parsing the command-line input and initializing the wave function are straightforward and implemented in the usual way.

Calculation of the advanced wave function and the time propagator matrix is done according to the stencil given in Fig. 3. The resulting system of linear equations is solved using a banded matrix solver, namely the zbgsv routine of the LAPACK package [1]. The storage scheme is arranged in such a way that the bandwidth of the resulting matrix depends upon the level in y-direction as this level is normally smaller than the level in x-direction.

After the wave function is updated, both its l_2-norm,

$$\|\psi_{ij}\|_2 = \left((n_x n_y)^{-1} \sum_{ij} \psi_{ij}\psi_{ij}^*\right)^{1/2}, \qquad (6)$$

and its quantum mechanical integral norm,

$$\|\psi\| = \iint |\psi|^2 dx\, dy, \tag{7}$$

are calculated to check the convergence of the approximation. The integral norm is computed with the extended trapezoidal rule,

$$\int_a^b f(x)dx \approx h\left(\frac{1}{2}f_1 + f_2 + \cdots + f_{N-1} + \frac{1}{2}f_N\right),$$

applied to our two-dimensional integral [16]. Note that (7) is calculated in the untransformed cartesian coordinates. In the transformed curvilinear coordinates it reads

$$\|\psi\| = \iint |\psi|^2 \left|\frac{\partial(x,y)}{\partial(u,v)}\right| du\, dv. \tag{8}$$

4 Results

In Fig. 4 and 6 we show the evolution of a Gaussian wave packet unconfined in space and one in a box with hard wall boundary conditions. The y-dimension has been suppressed for these calculations, and the box simulation was done by setting the metric to the identity metric. Fig. 4 shows the spreading of a Gaussian wave packet centered at the middle of the box with zero momentum. In the last three pictures, the reflection from the hard wall boundaries makes the two cases already considerably different.

To test the validity of our FD approximation, one can check whether the obtained discrete solutions obey a Richardson expansion [4]. If we define

$$Q = \frac{\|\psi^{4h} - \psi^{2h}\|_2}{\|\psi^{2h} - \psi^{h}\|_2}, \tag{9}$$

then in the limit of $h \to 0$ we should get

$$\lim_{h\to 0} Q = 4 \tag{10}$$

In Fig. 5, values of Q for both open and hard wall boundaries are plotted for various grid levels. It shows clearly that our solution indeed passes this test.

In Fig. 6 we show a Gaussian wave packet with initial momentum to the right. Whereas in the open boundary simulation the wave packet just leaves our plotting region, it is being reflected for the hard wall case. After the center of the wave packet has moved again away from the right boundary and the interference effects have decreased, the initial Gaussian shape of the packet is restored and reappears.

Fig. 7 shows the reflection and transmission of a wave packet at a potential barrier. The wave packet again has an initial momentum to the right, and as soon as the wave packet hits the left boundary of the barrier, part of it is being reflected. The same happens again on the right boundary of the barrier resulting in two reflected wave packets and one transmitted wave packet. Higher order reflections are too small to be observed. Furthermore, the characteristic discontinuities in the derivative of the wave function due to the jump in the potential are clearly visible during the wave packet's transmission.

For simulations with a large final time, the wave packet will eventually reach one of the boundaries of our computational domain. Although the computational domain is mapped onto an infinite physical domain, the simulation will only be accurate for small amplitudes and small momenta in the outer regions of our domain. Using the transformation given in section 2, the uniform grid points at the borders are effectively mapped onto a grid that gets coarser and coarser towards the borders. As soon as a wave packet with considerable amplitude/momentum moves into this region, the simulation can no longer be accurate. The wave packet will be partially reflected and destroyed. This is depicted in Fig. 8. Nevertheless, in contrast to a hard wall boundary, the reflection happens at a later time and more slowly.

Finally, to simulate a wave packet in a wave guide we use the transformation discussed in section 2. The resulting grid for a simulation with level 10 in x-direction, level 5 in y-direction is shown in Fig. 9. Fig. 10 shows the initial condition, a Gaussian wave packet in its second transversal mode. Fig. 11 and 12 show the evolution of the wave packet. The reflected parts of the wave packet are only barely visible in Fig. 12.

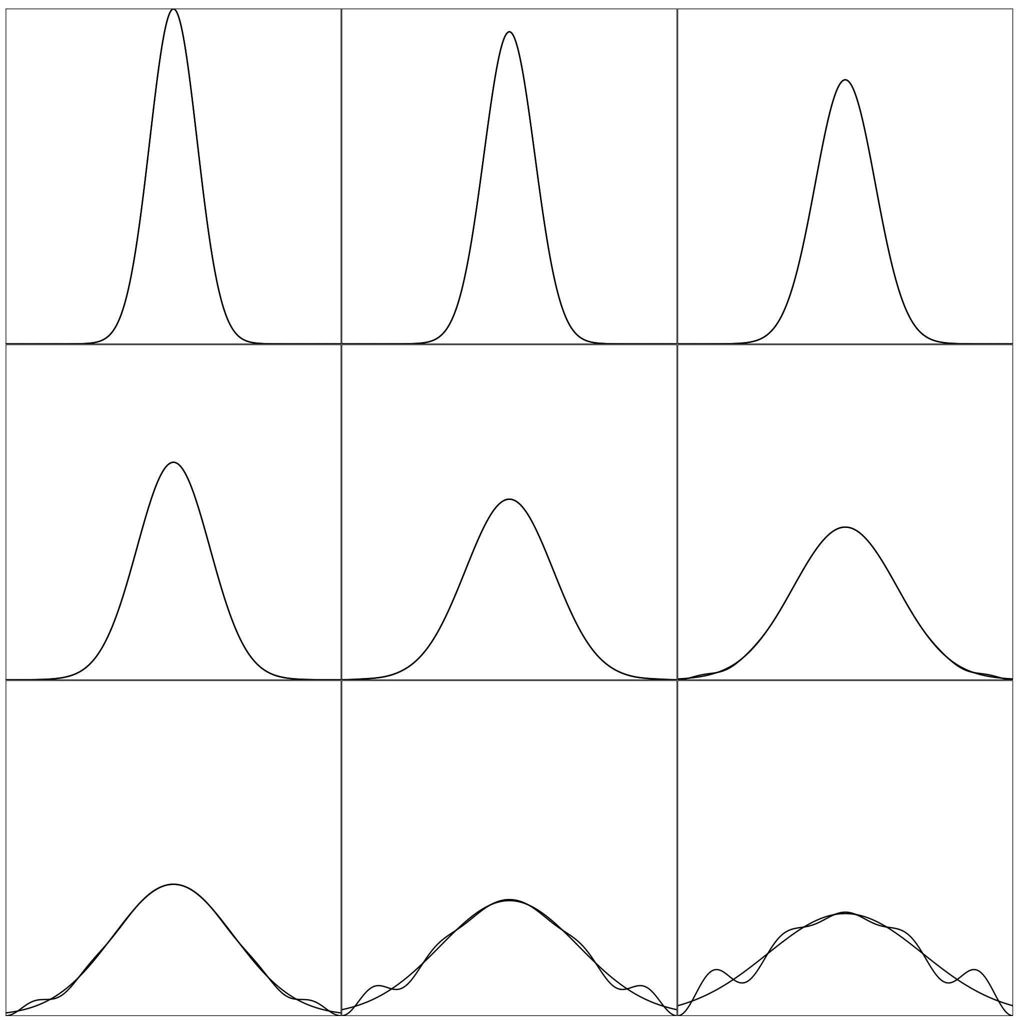

Figure 4: Spreading of a Gaussian wave packet

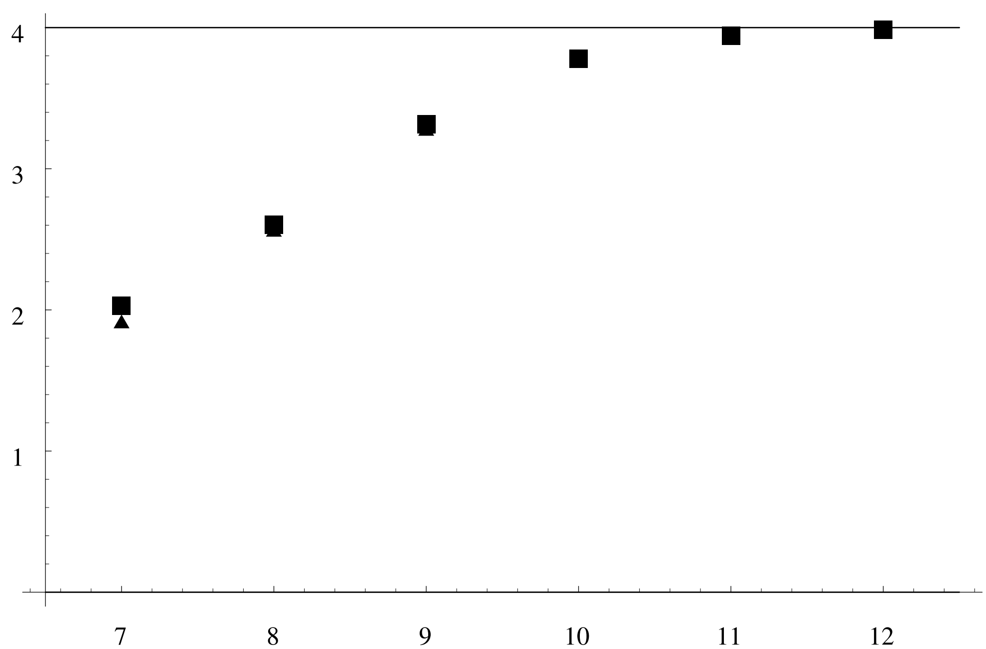

Figure 5: Richardson extrapolation

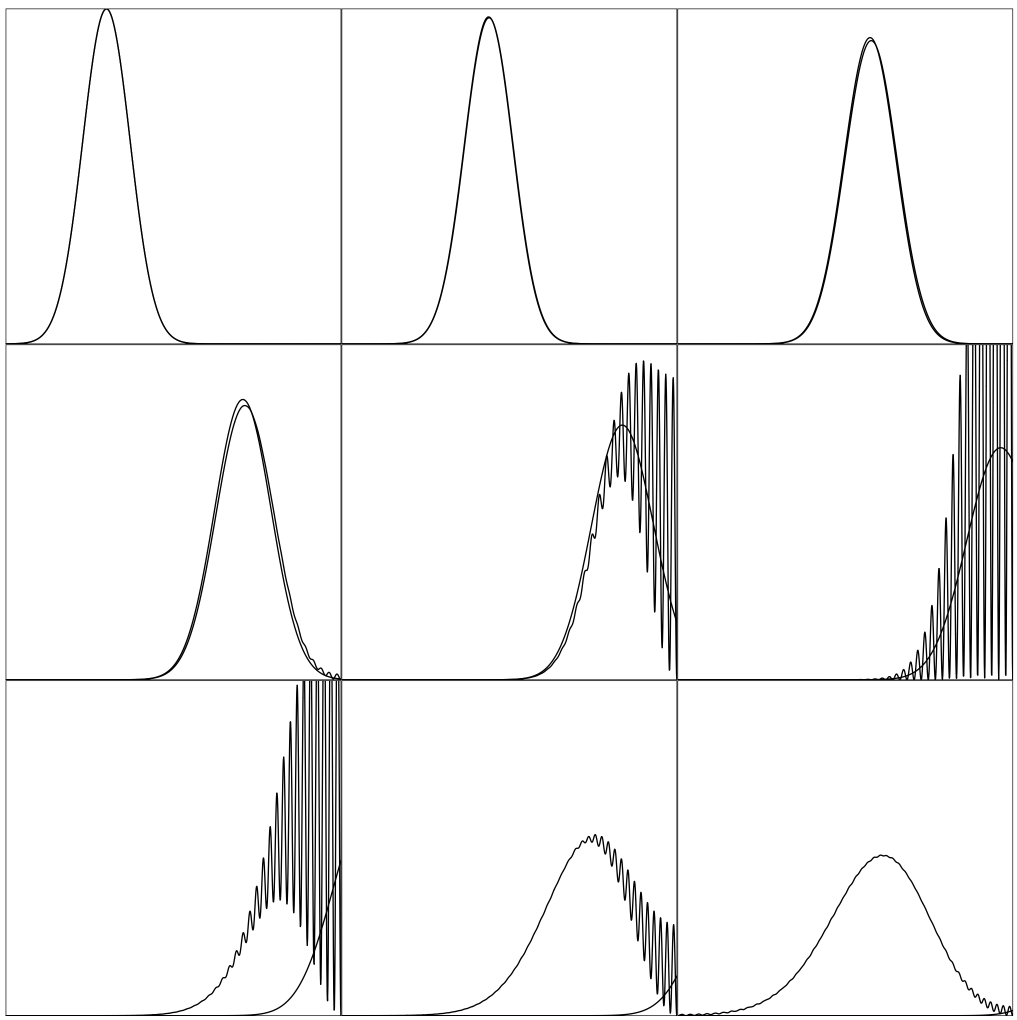

Figure 6: Reflection of a Gaussian wave packet

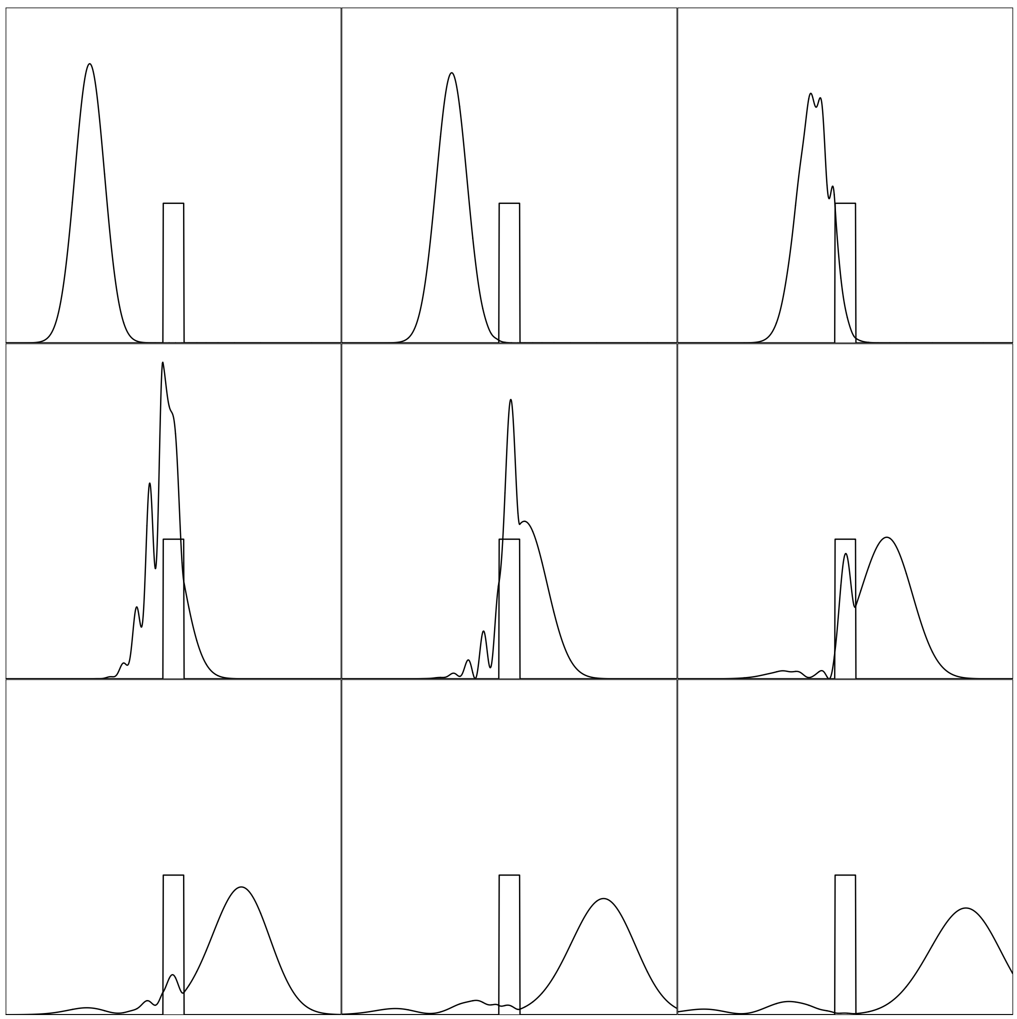

Figure 7: Reflection and transmission at a potential barrier

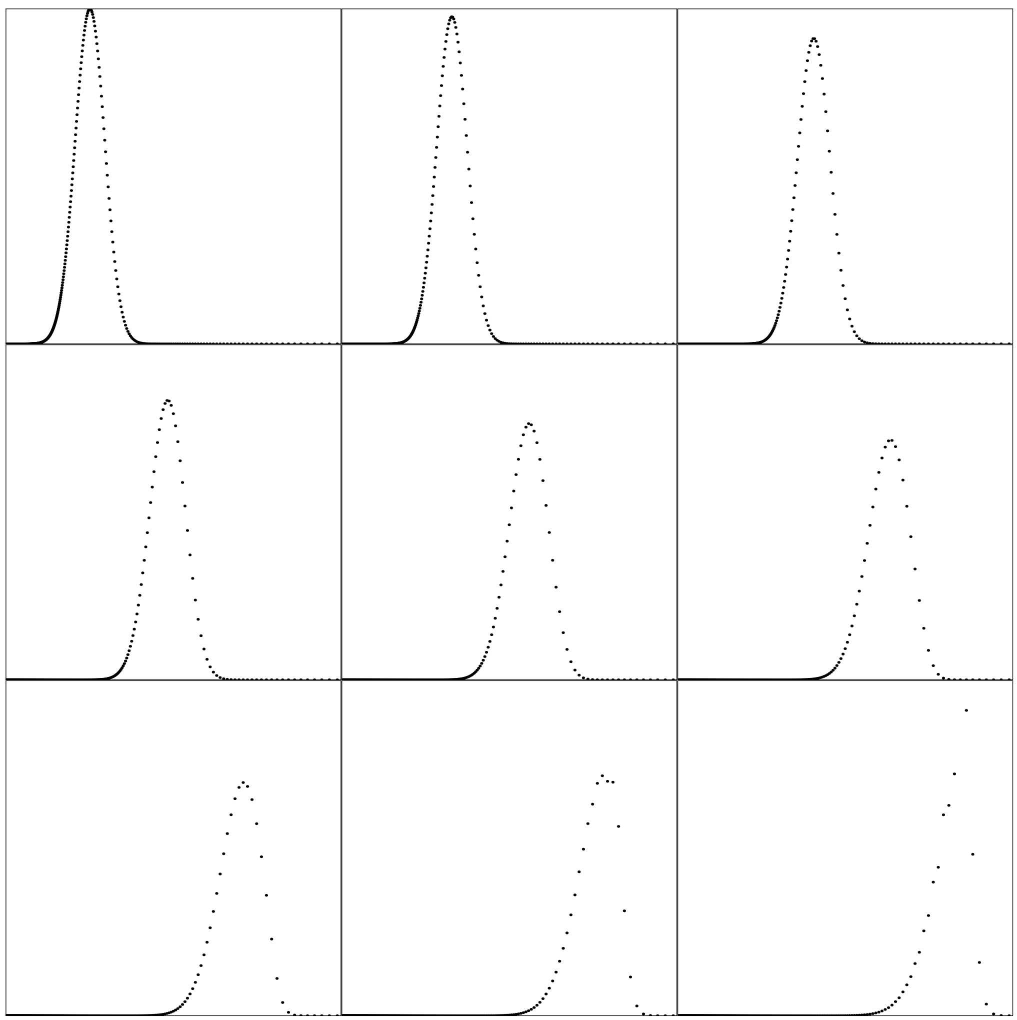

Figure 8: Reflection at the "open" boundary

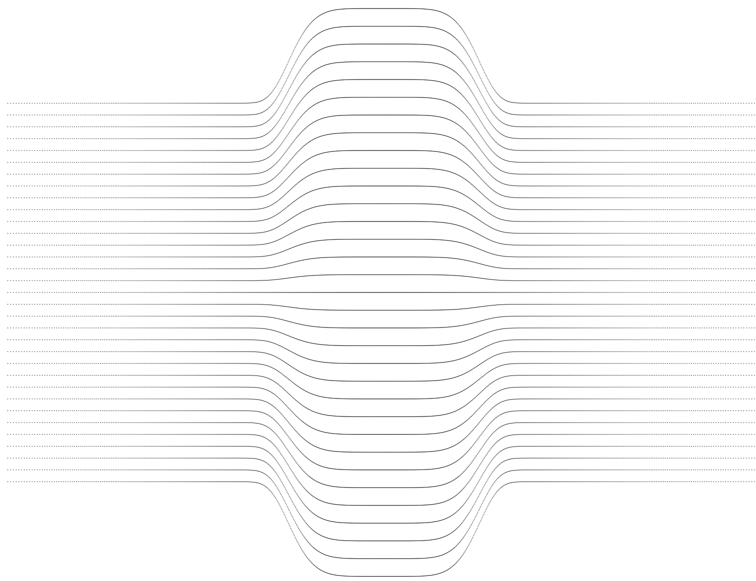

Figure 9: Grid for waveguide simulation

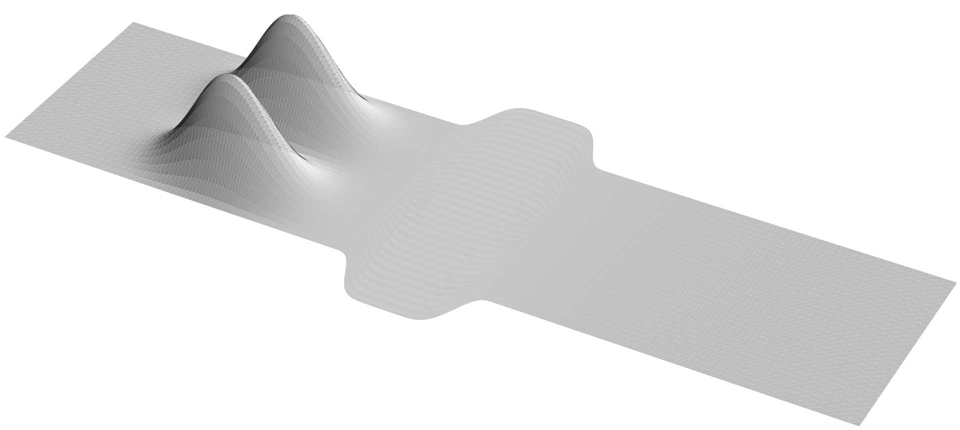

Figure 10: Wave packet incident from left on the cavity

Figure 11: Wave packet being scattered in the cavity

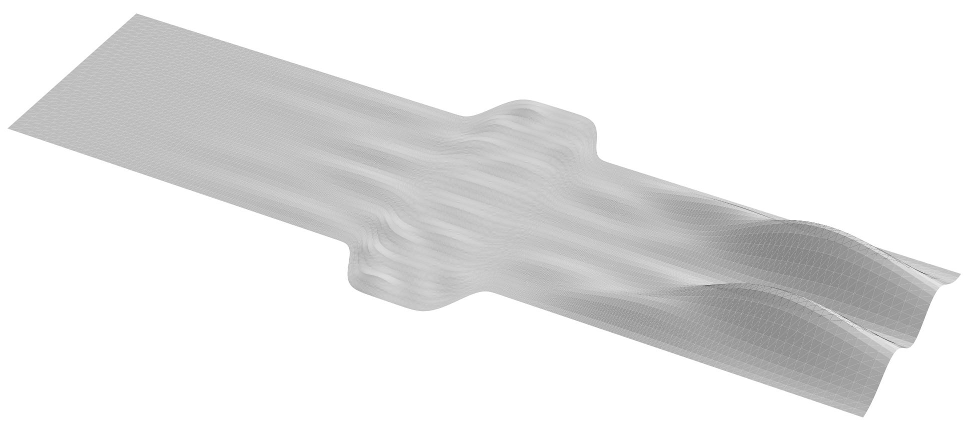

Figure 12: Transmitted wave packet

5 Conclusion

In this report we have shown how to simulate the two-dimensional Schrödinger equation for an arbitrary metric and applied this to an electron waveguide. The presented approach is still rather rudimentary, and a lot is left for improvement. Most important improvements are

- a better simulation of open boundary conditions, i.e. implementation of outgoing wave boundary conditions or annihilating boundary layers.
- solving the Crank-Nicholson system of linear equations more efficiently with respect to time and memory, e.g. by implementing an iterative multigrid algorithm.

Nevertheless, the chosen approach proved to be quite feasible.

References

[1] E. Anderson et al. LAPACK user's guide. http://www.netlib.org/lapack/lug/lapack_lug.html, 1994.

[2] Thomas B. Bahder. *Mathematica for Scientists and Engineers.* Addison-Wesley, 1995.

[3] Ilja N. Bronstein et al. *Taschenbuch der Mathematik.* Verlag Harri Deutsch, 1993.

[4] Matthew W. Choptuik. Physics 381C: Computational physics. Course notes. http://wwwrel.ph.utexas.edu/Members/matt/Teaching/97Spring/Phy381C/Notes.html, 1997.

[5] Akira Endoh, Shigehiko Sasa, Hiroshi Arimoto, and Shunichi Moto. Dynamics of electron-wave packets in nanostructures. In *Inst. Phys. Conf. Ser. No. 127*, pages 33 – 36, 1992.

[6] Abraham Goldberg, Harry M. Schey, and Judah L. Schwartz. Computer-generated motion pictures of one-dimensional quantum-mechanical transmission and reflection phenomena. *Am. J. Phys.*, 35(3):177 – 186, 1967.

[7] W. Greiner. *Quantum Mechanics. An Introduction.* Springer, 1989.

[8] Reid Larimore Guenther. *A numerical study of the time dependent Schrödinger equation coupled with Newtonian gravity.* PhD thesis, The University of Texas at Austin, 1995.

[9] Harro Heuser. *Funktionalanalysis. Theorie und Anwendung.* B. G. Teubner, 3rd edition, 1992.

[10] David J. Kirkner, Craig S. Lent, and Srinivas Sivaprakasam. The numerical simulation of electron transmission through a two-dimensional quantum device by the finite element method. *Int. J. Numer. Methods Eng.*, 29:1527 – 1537, 1990.

[11] R. Landauer. *IBM J. Res. Dev.*, 1:223, 1957.

[12] Craig S. Lent and David J. Kirkner. The quantum transmitting boundary method. *J. Appl. Phys.*, 67(10):6353 – 6359, 1990.

[13] Eugen Merzbacher. *Quantum Mechanics.* John Wiley & Sons, 2nd edition, 1970.

[14] Kyungsun Na and L.E. Reichl. The intrinsic electron lifetimes in a ballistic electron waveguide. To be published.

[15] Clive G. Page. Professional programmer's guide to fortran 77. ftp://ftp.star.le.ac.uk/pub/fortran/prof77.tex.gz, 1995.

[16] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C. The Art of Scientific Computing.* Cambridge University Press, 2nd edition, 1992.

[17] Leonard F. Register, Umberto Ravaioli, and Karl Hess. Numerical simulation of mesoscopic systems with open boundaries using the multidimensional time-dependent schrödinger equation. *J. Appl. Phys.*, 69:7153 – 7158, 1991. Cf. Erratum in *J. Appl. Phys.*, 71(3):1555, 1992.

[18] Fernando Sols, M. Macucci, U. Ravaioli, and Karl Hess. On the possibility of transistor action based on quantum interference phenomena. *Appl. Phys. Lett.*, 54(4):350 – 352, 1989.

[19] Fernando Sols, M. Macucci, U. Ravaioli, and Karl Hess. Theory for a quantum modulated transistor. *J. Appl. Phys.*, 66(8):3892 – 3906, 1989.

A Program listings

A.1 wg.f

```
c===========================================================
c
c     Term project for PHY 381C: Computational Physics
c     Markus Lischka, Spring 1997
c
c     Numerical solution of the time-dependent Schroedinger
c     equation for a 2D electron waveguide
c
c
c     Implementation details:
c     cf. term paper
c
c     Command line arguments:
c       usage: wg <tfinal> <levelx> <levely> <olevel> <dt/dx>
c                  <k> <x0> <sigma0> <c1> [<c2> ...]
c
c     Output on stdout:
c     1. Grid mesh
c     2. Solution at every 2**(levelx-olevel)'th time step
c
c     Output on stderr:
c     Depends on the setting of tracing variables in fcn.inc
c
c===========================================================

      program wg

      implicit none

c-----------------------------------------------------------
c     Variables shared by all routines
c-----------------------------------------------------------
      include       'fcn.inc'

      real*8         tfinal, dtdu, k, x0, s0, hu, hv
      integer        nu, nv, n, output

      real*8         c(maxcoeff)

c-----------------------------------------------------------
c     Note: The program makes use of the FORTRAN storage
c     scheme for arrays, i.e. it relies on
c       psi(j,i) = psi(j+nv*(i-1))
c     where i and j label the u- and v-direction
c     respectively. This is necessary to reduce the band-
c     width of the time propagator matrix (it is assumed
c     that the grid in v-direction is usually coarser than
c     in u-direction).
c     Furthermore, the array psi is always passed to
c     subroutines with dimension (nv,nu), NOT (maxnv, maxnu)
c-----------------------------------------------------------
      real*8         grid(2,maxnv,maxnu)
```

```
      complex*16     psi(maxnv,maxnu)


      call parsing(tfinal, dtdu, k, x0, s0, c, n,
     &             hu, hv, nu, nv, output)
      call init(k, x0, s0, c, n, hu, hv, nu, nv,
     &          grid, psi)

      if( asciitrace ) then
         call writegrid(nu, nv, grid)
      end if

      call solve(tfinal, dtdu, hu, hv,
     &           nu, nv, output,
     &           grid, psi)

      if( gft_trace ) then
         call gft_close_all()
      end if

      stop
      end

c============================================================
c
c     Command line parsing
c
c     Uses Matt Choptuik's p329f routines
c
c============================================================

      subroutine parsing(tfinal, dtdu, k, x0, s0, c, n,
     &                   hu, hv, nu, nv, output)

      implicit       none

c------------------------------------------------------------
c     Read in common definitions
c------------------------------------------------------------
      include       'fcn.inc'

      real*8         tfinal, dtdu, k, x0, s0, hu, hv
      integer        n, nu, nv, output

      real*8         c(n)

c------------------------------------------------------------
c     Variables and routines for parsing input
c------------------------------------------------------------
      integer        levelu, levelv, olevel, params

      integer        iargc, i4arg
      real*8         r8arg

      real*8         r8_never
```

```
      parameter    ( r8_never = -1.0d-60 )

      integer        i4_never
      parameter    ( i4_never = 0 )

      character*2    cdnm
      parameter    ( cdnm = 'wg' )

c-----------------------------------------------------------
c     Variables for initialization
c-----------------------------------------------------------
      integer        i, iu, iv


c===========================================================
c     Start of subroutine
c===========================================================

c-----------------------------------------------------------
c     read in FDA parameters from command line
c-----------------------------------------------------------

      params = iargc()
      if( params .lt. 9 ) go to 900

      tfinal  = r8arg(1,r8_never)
      levelu  = i4arg(2,i4_never)
      levelv  = i4arg(3,i4_never)
      olevel  = i4arg(4,i4_never)
      dtdu    = r8arg(5,r8_never)
      k       = r8arg(6,r8_never)
      x0      = r8arg(7,r8_never)
      s0      = r8arg(8,r8_never)

      if( tfinal .eq. r8_never .or. levelu .le. i4_never
     &    .or. levelv .le. i4_never .or. olevel .le. i4_never
     &    .or. dtdu .eq. r8_never   .or. k .eq. r8_never
     &    .or. x0 .eq. r8_never     .or. s0 .eq. r8_never )
     &   go to 900

      if( inputtrace ) then
         write(0,*) '<<< wg: command line input >>>'
         write(0,*) 'tfinal = ', tfinal
         write(0,*) 'levelu = ', levelu
         write(0,*) 'levelv = ', levelv
         write(0,*) 'olevel = ', olevel
         write(0,*) 'dtdu   = ', dtdu
         write(0,*) 'k      = ', k
         write(0,*) 'x0     = ', x0
         write(0,*) 's0     = ', s0
      end if

c-----------------------------------------------------------
c     set up FDA parameters
c-----------------------------------------------------------
```

```
      nu = 2**levelu + 1
      nv = 2**levelv + 1

      if( nu .gt. maxnu .or. nv .gt. maxnv ) then
         write(0,*) cdnm//': Not enough memory: level too high!'
         stop
      end if

      hu = 1.0d0/(nu-1)
      hv = 1.0d0/(nv-1)

      output = 2**(levelu - olevel)

c------------------------------------------------------------
c     read in coefficients of transversal modes
c------------------------------------------------------------
      n = params - 8
      do i = 1, min(n, maxcoeff)
         c(i) = r8arg(i+8,r8_never)
         if( c(i) .eq. r8_never ) go to 900
         if( inputtrace ) then
            write(0,*) 'c(',i,') = ', c(i)
         end if
      end do

      return

900   continue
      write(0,*) 'usage: wg <tfinal> '//
     &           '<levelx> <levely> <olevel> <dt/dx>'
      write(0,*) '          <k> <x0> <sigma0> <c1> [<c2> ...]'
      stop

      end


c============================================================
c
c     Initialization of wave function and grid
c
c============================================================

      subroutine init(k, x0, s0, c, n, hu, hv, nu, nv,
     &                grid, psi)

      implicit       none

c------------------------------------------------------------
c     Read in common definitions
c------------------------------------------------------------
      include       'fcn.inc'

      real*8         k, x0, s0, hu, hv
      integer        nu, nv, n

      real*8         c(n)
```

```
      real*8         grid(2,nv,nu)
      complex*16     psi(nv,nu)

c------------------------------------------------------------
c     Locals
c------------------------------------------------------------
      integer        i, iu, iv

c------------------------------------------------------------
c     initialize solution with transversal modes
c------------------------------------------------------------
      do iu = 1, nu
         do iv = 1, nv
            grid(1,iv,iu) = xct((iu-1)*hu,(iv-1)*hv)
            grid(2,iv,iu) = yct((iu-1)*hu,(iv-1)*hv)
            psi(iv,iu) = (0.0d0, 0.0d0)
         end do
      end do

      do i = 1, n
         do iu = 1, nu
            do iv = 1, nv
               psi(iv,iu) = psi(iv,iu) +
     &              c(i) * sin( i*pi*grid(2,iv,iu) )
            end do
         end do
      end do

c------------------------------------------------------------
c     for debugging purposes: initialize with sinusoidal
c     longitudinal mode (stationary state for trivial metric)
c------------------------------------------------------------
c      do iu = 1, nu
c         do iv = 1, nv
c            psi(iv,iu) = psi(iv,iu) *
c     &          sin(pi*grid(1,iv,iu))
c         end do
c      end do

c------------------------------------------------------------
c     initialize solution with longitudinal gauss wave packet
c------------------------------------------------------------
      do iu = 1, nu
         do iv = 1, nv
            psi(iv,iu) = psi(iv,iu) * exp(
     &           dcmplx(
     &              -(grid(1,iv,iu) - x0)**2 / (2 * s0**2),
     &              k * grid(1,iv,iu)
     &           ) )
         end do
      end do

c------------------------------------------------------------
c     enforce boundary conditions
c------------------------------------------------------------
      do iu = 1, nu
```

```
         psi(1,iu)  = (0.0d0,0.0d0)
         psi(nv,iu) = (0.0d0,0.0d0)
      end do
      do iv = 1, nv
         psi(iv,1)  = (0.0d0,0.0d0)
         psi(iv,nu) = (0.0d0,0.0d0)
      end do

      return
      end


c============================================================
c
c     Solve Schroedinger equation
c
c============================================================
      subroutine solve(tfinal, dtdu, hu, hv,
     &           nu, nv, output,
     &           grid, psi)

      implicit       none
c------------------------------------------------------------
c     Read in common definitions
c------------------------------------------------------------
      include       'fcn.inc'

      real*8         tfinal, dtdu, hu, hv
      integer        nu, nv, output

      real*8         grid(2,nv,nu)
      complex*16     psi(nv,nu)

c------------------------------------------------------------
c     Variables for LAPACK routine zgbsv
c------------------------------------------------------------
      integer        nrhs
      parameter    ( nrhs = 1 )

      integer        n, kl, ku, ldab, ldb, info
      integer        ipiv(maxnu*maxnv)

      complex*16     ab(2*(maxnv+1)+(maxnv+1)+1, maxnu*maxnv)

c------------------------------------------------------------
c     Local variables
c------------------------------------------------------------
      complex*16     row(-maxnv-1:maxnv+1)
      real*8         dt
      integer        tstep, tstepmax

      integer        iu, iv, i, diag, ri, ci

      if( routinetrace ) then
         write (0,*) '<<< Solve Schroedinger equation >>>'
```

```
      end if

      call writesol(0.0d0,nu,nv,psi,grid)

c------------------------------------------------------------
c     set up constant loop parameters
c------------------------------------------------------------
      dt = dtdu*hu
      tstepmax = tfinal/dt

c     don't calculate steps after last output
      tstepmax = tstepmax - mod(tstepmax, output)

      n = nu*nv
      kl = nv+1
      ku = nv+1
      ldab = 2*(maxnv+1)+(maxnv+1)+1
      ldb  = n

      do tstep = 1, tstepmax
         if( routinetrace ) then
            write (0,*) ' Timestep ', tstep*dtdu*hu
         end if

         call lop(psi, nu, nv, hu, hv,
     &            tstep*dtdu*hu, 0.5d0*dtdu*hu)

c------------------------------------------------------------
c     set up banded matrix
c------------------------------------------------------------
         do iu = 1, nu
            do iv = 1, nv
               call lopr(row, iu, iv, nu, nv,
     &                   hu, hv, tstep*dtdu*hu, -0.5d0*dtdu*hu)
               diag = iv + nv*(iu-1)
               do i = max(-diag+1,-nv-1), min(nu*nv-diag,nv+1)
                  ab(kl+ku+1 -i,diag+i) = row(i)
               end do
            end do
         end do

         call zgbsv(n, kl, ku, nrhs, ab, ldab, ipiv, psi,
     &              ldb, info)

         if( normtrace ) then
            call writenorm(tstep*dt,nu,nv,hu,hv,psi)
         end if

         if( mod(tstep, output) .eq. 0 ) then
            call writesol(tstep*dt,nu,nv,psi,grid)
         end if
      end do

      return
      end
```

```
c===========================================================
c
c     Write grid to stdout
c
c===========================================================

      subroutine writegrid(nu, nv, grid)

      implicit       none
c-----------------------------------------------------------
c     Read in common definitions
c-----------------------------------------------------------
      include       'fcn.inc'

      integer        nu, nv
      real*8         grid(2,nv,nu)

      integer        iu, iv

      if( routinetrace ) then
         write (0,*) '<<< Output grid >>>'
      end if
      write (*,*) nu, nv
      do iu = 1, nu
         do iv = 1, nv
            write (*,*) grid(1,iv,iu), grid(2,iv,iu)
         end do
      end do

      return
      end

c===========================================================
c
c     Write solution to stdout
c
c===========================================================

      subroutine writesol(t, nu, nv, psi, grid)

      implicit       none
c-----------------------------------------------------------
c     Read in common definitions
c-----------------------------------------------------------
      include       'fcn.inc'

      real*8         t
      integer        nu, nv

      complex*16     psi(nv,nu)
      real*8         grid(2,nv,nu)

c-----------------------------------------------------------
c     Definitions for gft_trace output using BBH routines
c-----------------------------------------------------------
```

```
      real*8         bbox(4), psiabs(maxnv*maxnu)
      integer        shape(2), rank

c-----------------------------------------------------------
c     Locals
c-----------------------------------------------------------
      integer        iu, iv

      if( routinetrace ) then
         write (0,*) '<<< Output solution >>>'
      end if

      if( asciitrace ) then
         write (*,*) t
         do iu = 1, nu
            do iv = 1, nv
               write (*,100) psi(iv,iu)
 100           format('(',E14.7E3,',',E14.7E3,')')
            end do
         end do
      end if

c-----------------------------------------------------------
c     gft output if requested
c-----------------------------------------------------------
      if( gft_trace ) then
         bbox(1) = 0.0d0
         bbox(2) = 1.0d0
         bbox(3) = 0.0d0
         bbox(4) = 1.0d0

         rank = 2

         shape(1) = nv
         shape(2) = nu

         do iu = 1, nu
            do iv = 1, nv
               psiabs(iv+nv*(iu-1))=abs(psi(iv,iu))**2
            end do
         end do

         call gft_write_bbox('wg',t,shape,rank,bbox,psiabs)
      end if

      if( ser_trace ) then
         call writesols(t,nu,nv,psi,grid)
      end if

      return
      end

c===========================================================
c
c     Write solution at y = 1/2 to ser
c
```

```
c===========================================================

      subroutine writesols(t, nu, nv, psi, grid)

      implicit       none
c-----------------------------------------------------------
c     Read in common definitions
c-----------------------------------------------------------
      include       'fcn.inc'

      real*8         t
      integer        nu, nv

      complex*16     psi(nv,nu)
      real*8         grid(2,nv,nu)

c-----------------------------------------------------------
c     Definitions for ser
c-----------------------------------------------------------
      real*8         x(maxnu), psiabs(maxnu)

c-----------------------------------------------------------
c     Locals
c-----------------------------------------------------------
      integer        iu

      if( routinetrace ) then
         write (0,*) '<<< Output solution to ser >>>'
      end if

      do iu = 1, nu
         x(iu)=grid(1,(nv-1)/2+1,iu)
         psiabs(iu)=abs(psi((nv-1)/2+1,iu))**2
      end do

      call vsxynt('abs2',t,x,psiabs,nu)

      return
      end

c===========================================================
c
c     Write norm to stderr
c
c     calculates both l2-norm of the solution vector and
c     the quantum mechanical norm of the wavefunction
c
c===========================================================

      subroutine writenorm(t, nu, nv, hu, hv, psi)

      implicit       none
c-----------------------------------------------------------
c     Read in common definitions
c-----------------------------------------------------------
      include       'fcn.inc'
```

```
      real*8         t, hu, hv
      integer        nu, nv

      complex*16     psi(nv,nu)

c------------------------------------------------------------
c     Locals
c------------------------------------------------------------
      integer        Iu, Iv
      real*8         norml2, norm, normv

      if( routinetrace ) then
         write (0,*) '<<< Output norm at t =', t, '>>>'
      end if

      norml2 = 0.0d0
      do iu = 1, nu
         do iv = 1, nv
            norml2 = norml2 + abs(psi(iv,iu))**2
         end do
      end do

c------------------------------------------------------------
c     two-dimensional integration of abs(psi)^2:
c     the boundaries are zero anyway, so they are not
c     integrated.
c------------------------------------------------------------

      norm = 0.0d0
      do iu = 2, nu-1
         normv = 0.0d0
         do iv = 2, nv-1
            normv = normv + abs(psi(iv,iu))**2 *
     &                     jd((iu-1)*hu,(iv-1)*hv)
         end do
         norm = norm + normv
      end do


      write(0,*) 'norm:', norm, ' l2-norm:', norml2

      return
      end
```

A.2 fcn.f

```
c===========================================================
c
c     Shared functions and subroutines for wg.f
c     related to the physical problem being solved
c
c===========================================================

c-----------------------------------------------------------
c     coordinate transforms (u,v)->(x,y)
c-----------------------------------------------------------
      function xct(u,v)

      implicit none
      include  'fcn.inc'
      real*8   u, v

c     identity transformation
c      xct = u

c     mapping (0,1) -> (-inf,inf) for waveguide
      xct = tan(pi*(u-0.5d0))

      return
      end


      function yct(u,v)

      implicit none
      include  'fcn.inc'
      real*8   u, v

c     identity transformation
c      yct = v

c     mapping cavity -> straight boundaries for waveguide
      yct = (w + h*xi(u))*(2.0d0*v-1.0d0)

      return
      end

c-----------------------------------------------------------
c     time and space-dependent potential
c-----------------------------------------------------------
      function pot(u,v,t)

      implicit none
      include  'fcn.inc'
      real*8   u, v, t

c     vacuum
      pot = 0.0d0

      return
      end
```

```
c---------------------------------------------------------
c     coefficients for derivatives derived from metric
c---------------------------------------------------------
      function a20(u,v)

      implicit none
      include  'fcn.inc'
      real*8   u, v

c     identity transformation
c      a20 = 1.0d0

c     mapping (0,1) -> (-inf,inf), cavity -> straight boundaries
      a20 = sin(pi*u)**4 / pi**2

      return
      end


      function a02(u,v)

      implicit none
      include  'fcn.inc'
      real*8   u, v

c     identity transformation
c      a02 = 1.0d0

c     mapping (0,1) -> (-inf,inf), cavity -> straight boundaries
      a02 = ((pi**2 + h**2*(1 - 2*v)**2*sin(pi*u)**4*
     &       xiprime(u)**2))/
     &  (4.*pi**2*(w + h*xi(u))**2)

      return
      end


      function a11(u,v)

      implicit none
      include  'fcn.inc'
      real*8   u, v

c     identity transformation
c      a11 = 0.0d0

c     mapping (0,1) -> (-inf,inf), cavity -> straight boundaries
      a11 = -((h*(-1 + 2*v)*sin(pi*u)**4*xiprime(u)
     &      )/(pi**2*(w + h*xi(u))))

      return
      end


      function a10(u,v)
```

```
      implicit none
      include  'fcn.inc'
      real*8   u, v

c     identity transformation
c      a10 = 0.0d0

c     mapping (0,1) -> (-inf,inf), cavity -> straight boundaries
      a10 = (2*cos(pi*u)*sin(pi*u)**3)/pi

      return
      end


      function a01(u,v)

      implicit none
      include  'fcn.inc'
      real*8   u, v

c     identity transformation
c      a01 = 0.0d0

c     mapping (0,1) -> (-inf,inf), cavity -> straight boundaries
      a01 = -(h*(-1 + 2*v)*sin(pi*u)**3*
     &     (2*pi*cos(pi*u)*(w + h*xi(u))*xiprime(u) -
     &       2*h*sin(pi*u)*xiprime(u)**2 +
     &       sin(pi*u)*(w + h*xi(u))*xiprime2(u))
     &     )/(2.*pi**2*(w + h*xi(u))**2)

      return
      end

c-----------------------------------------------------------
c     Jacobi determinant for integration
c-----------------------------------------------------------
      function jd(u,v)

      implicit none
      include  'fcn.inc'
      real*8   u,v

c     identity transformation
c      jd = 1.0d0

c     mapping (0,1) -> (-inf,inf), cavity -> straight boundaries
      jd = pi * (w + h*xi(u)) / sin(pi*u)**2

      return
      end

c-----------------------------------------------------------
c     support functions for describing cavity
c-----------------------------------------------------------
      function xi(u)
```

```
      implicit none
      include  'fcn.inc'
      real*8   u

      xi = exp(-0.5d0 * tan(pi*(u-0.5d0))**(2*p) / sigma**2)

      return
      end


      function xiprime(u)

      implicit none
      include  'fcn.inc'
      real*8   u

      xiprime =
     &   p * pi * exp(-0.5d0/(tan(pi*u)**(2*p) * sigma**2)) /
     &   (sigma**2 * tan(pi*u)**(2*p)*sin(pi*u)*cos(pi*u))

      return
      end


      function xiprime2(u)

      implicit none
      include  'fcn.inc'
      real*8   u

      xiprime2 =
     &   p*pi**2*(-(sigma**2*cos(2*pi*u)) +
     &      p*(-2*sigma**2 + tan(pi*u)**(-2*p)))*
     &    (tan(pi*u)**p*sin(pi*u)*cos(pi*u))**(-2)/
     &  (exp(tan(pi*u)**(-2*p)/(2*sigma**2))*sigma**4)

      return
      end

c============================================================
c     Operator definition
c============================================================

      subroutine lopr (row, i, j,
     &                 nu, nv, hu, hv, t, dt)

      implicit none
c------------------------------------------------------------
c     lopr calculates the row of the banded matrix of the
c     time propagator corresponding to the grid point (i,j)
c------------------------------------------------------------
      include     'fcn.inc'

      complex*16  row(-maxnv-1:maxnv+1)
```

```
      real*8      t, dt, hu, hv
      integer     i, j, nu, nv

c-------------------------------------------------------------
c     Locals
c-------------------------------------------------------------
      integer     il
      real*8      u, v

c-------------------------------------------------------------
c     Assume the following storage scheme
c       psi(j,i) = psi(j+nv*(i-1))
c     for the conversion from 2d wavefunction to 1d vector!
c     row(0) indexes the diagonal element L(ri, ri) of the
c     time propagation operator with ri = j+nv*i.
c
c     Cf. term paper for applied differencing scheme.
c-------------------------------------------------------------
      u = (i-1)*hu
      v = (j-1)*hv

      if( i .eq. 1 .or. i .eq. nu .or.
     &    j .eq. 1 .or. j .eq. nv ) then
         do il = -nv-1, 1
            row(il) = (0.0d0, 0.0d0)
         end do
         row(0) = (1.0d0, 0.0d0)
         do il = 1, nv+1
            row(il) = (0.0d0, 0.0d0)
         end do
      else
         row(-nv-1) = dcmplx( 0.0d0, -dt*(
     &         -a11(u,v)/(8.0d0*hu*hv) ) )
         row(-nv)   = dcmplx( 0.0d0, -dt*(
     &         -a20(u,v)/(2.0d0*hu**2)
     &         +a10(u,v)/(4.0d0*hu) ) )
         row(-nv+1) = dcmplx( 0.0d0, -dt*(
     &         a11(u,v)/(8.0d0*hu*hv) ) )
         do il = -nv+2, -2
            row(il) = (0.0d0,0.0d0)
         end do
         row(-1)    = dcmplx( 0.0d0, -dt*(
     &         -a02(u,v)/(2.0d0*hv**2)
     &         +a01(u,v)/(4.0d0*hv) ) )
         row(0)     = dcmplx( 1.0d0, -dt*(
     &          a02(u,v)/(hv**2)
     &         +a20(u,v)/(hu**2)
     &         +pot(u,v,t+dt/2) ) )
         row(1)     = dcmplx( 0.0d0, -dt*(
     &         -a02(u,v)/(2.0d0*hv**2)
     &         -a01(u,v)/(4.0d0*hv) ) )
         do il = 2, nv-2
            row(il) = (0.0d0,0.0d0)
         end do
         row(nv-1)  = dcmplx( 0.0d0, -dt*(
     &          a11(u,v)/(8.0d0*hu*hv) ) )
```

```
         row(nv)    = dcmplx( 0.0d0, -dt*(
     &          -a20(u,v)/(2.0d0*hu**2)
     &          -a10(u,v)/(4.0d0*hu) ) )
         row(nv+1)  = dcmplx( 0.0d0, -dt*(
     &          -a11(u,v)/(8.0d0*hu*hv) ) )
      end if

      return
      end


      subroutine lop (psi, nu, nv,
     &                hu, hv, t, dt)

      implicit none
c------------------------------------------------------------
c     lop applies the time propagator e^(- i H t) to psi
c     and returns the result in psi
c------------------------------------------------------------
      include 'fcn.inc'

      complex*16  psi(nv,nu)
      integer     nu, nv, diag
      real*8      hu, hv, t, dt

      complex*16  row(-maxnv-1:maxnv+1)
      complex*16  psi2(maxnu*maxnv)

      integer     iv, iu, k

      do iu = 1, nu
         do iv = 1, nv
            psi2(iv+nv*(iu-1)) = psi(iv,iu)
         end do
      end do

      do iu = 1, nu
         do iv = 1, nv
            call lopr(row, iu, iv, nu, nv, hu, hv, t, dt)
            psi(iv,iu) = (0.0d0,0.0d0)
            diag = iv + nv*(iu-1)
            do k = max(-diag+1,-nv-1), min(nu*nv-diag,nv+1)
               psi(iv,iu) = psi(iv,iu) + row(k)*psi2(iv+nv*(iu-1)+k)
            end do
         end do
      end do

      return
      end
```

A.3 fcn.inc

```
c=============================================================
c
c     Shared definitions for wg.f
c
c=============================================================

c-------------------------------------------------------------
c     Array sizes
c-------------------------------------------------------------
      integer     maxnu, maxnv
      parameter ( maxnu = 2**10 + 1, maxnv = 2**5 + 1 )

      integer     maxcoeff
      parameter ( maxcoeff = 5 )

c-------------------------------------------------------------
c     Constants
c-------------------------------------------------------------
      real*8      pi
      parameter ( pi = 3.141 5926 5358 9793 d0 )

      logical     routinetrace
      parameter ( routinetrace = .true. )

      logical     inputtrace
      parameter ( inputtrace = .false. )

      logical     normtrace
      parameter ( normtrace = .true. )

c-------------------------------------------------------------
c     Parameters determining output
c-------------------------------------------------------------
      logical     gft_trace
      parameter ( gft_trace = .true. )

      logical     asciitrace
      parameter ( asciitrace = .true. )

      logical     ser_trace
      parameter ( ser_trace = .false. )

c-------------------------------------------------------------
c     Functions for coordinate transform (u,v) -> (x,y)
c-------------------------------------------------------------
      real*8   xct, yct

c-------------------------------------------------------------
c     Potential function
c-------------------------------------------------------------
      real*8      pot

c-------------------------------------------------------------
c     Coefficient functions for derivatives
c-------------------------------------------------------------
```

```
      real*8      a20, a02, a11, a10, a01

c------------------------------------------------------------
c     Jacobi determinant for integration
c------------------------------------------------------------
      real*8      jd

c------------------------------------------------------------
c     Support functions and constants for transformation
c------------------------------------------------------------
      real*8      xi, xiprime, xiprime2

      integer     p
      parameter ( p = 3 )

      real*8      h, w, sigma
      parameter ( h = 0.5d0 )
      parameter ( w = 1.0d0 )
      parameter ( sigma = 0.1d0 )
```

A.4 Makefile on the SGIs

```
############################################################
#  Note that this 'Makefile' assumes that the following
#  environment variables are set:
#
#     F77
#     F77FLAGS
#     F77CFLAGS
#     F77LFLAGS
#     LIBBLAS
#
#  Put the appropriate 'setenv' commands in your '~/.cshrc'.
#  See 'phy329@einstein:~/.cshrc' for an example.
############################################################
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS) -O
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)
LIBS         = -lp329f -llapack $(LIBBLAS)
UTILLIBS     = -lbbhutil -lmfhdf -ldf -ljpeg -lz -lvs


.f.o:
$(F77_COMPILE) $*.f

EXECUTABLES = wg

all: $(EXECUTABLES)

wg: wg.o fcn.o fcn.inc
$(F77_LOAD) wg.o fcn.o $(LIBS) $(UTILLIBS) -o wg

clean:
rm *.o
rm $(EXECUTABLES)
```